A MARSHALL CAVENDISH 6 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE

UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol 1 No 6

GAMES PROGRAMMING 6

CREATING THE BIG BANG 161

Routines you can add to a variety of games to create convincing screen 'explosions'

BASIC PROGRAMMING 11

SPRITES ON THE COMMODORE 64 168

A special, and important, graphics feature of these machines

BASIC PROGRAMMING 12

GET YOUR PROGRAMS IN SHAPE 173

The first of two articles explaining how to write better structured, smoother running programs

MACHINE CODE 7

GETTING DOWN BELOW ZERO 179

How to deal with negative numbers in hex and binary arithmetic – and when you'll need them

BASIC PROGRAMMING 13

REFINING YOUR SCREEN GRAPHICS 184

Getting to grips with more sophisticated uses of the BASIC graphics commands

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Dave King. Page 161, British Aerospace. Pages 162, 164, 166, Chris Lyons. Page 168, Tudor Art Studio/Bernard Robinson. Page 170, Bernard Fallon. Page 172, Howard Kingsnorth. Pages 174, 176, Nick Farmer. Page 178, Nick Farmer, Howard Kingsnorth. Page 180, Dave King. Page 182, David Lloyd, Dick Ward. Page 184, Hussein Hussein, Martin Cleaver. Page 186, Hussein Hussein, Peter Dazely. Page 188, David Lloyd. Page 190, Hussein Hussein, The Photographers Library.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland: Send £4.95 (inc p & p) (IR£5.45) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Gordon and Gotch Ltd, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Copies of any part of INPUT can be obtained from the following addresses at the regular cover price, with no extra charge for postage and packing:

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent

COPIES BY POST

Our Subscription Department can supply your copies direct to you regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. These rates apply anywhere in the world. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – **and please do not telephone.** Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +

COMMODORE 64 and 128

ACORN ELECTRON, BBC B and B+

DRAGON 32 and 64

ZX81

VIC 20

TANDY TRS80 COLOUR COMPUTER

CREATING THE BIG BANG

- CREATING SCREEN FLASHES
- A SIMPLE AEROPLANE BOMBING RUN PROGRAM
- HOW TO ADD FLAMES AND EXPLOSIONS

Explosions are a standard part of the games programmers' repertoire—in anything from aerial combat to interstellar wars. Here are some ways to program suitable visual effects that you can use for a whole variety of games.

It's quite easy to make your games a lot more spectacular by adding special effects graphics routines. And they need not be very complex programs to make a big difference to the look of your games.

As you'll see, there are many different ways to produce visual effects—all you have to remember is to use the right sort of effect in the right sort of game.

The flame effects detailed here are suitable for all sorts of games involving buildings, cars, ships, tanks and so on, but wouldn't really look right in a space game.

Also, as the flames and explosions have been drawn using UDGs, the maximum target size is limited. And this means that you can't just add them to any game, without altering the routine.

On the other hand, some of the machines have screen flash routines which are much better for space games, and have the added advantage that they need no adaptation for use in any game.

The key to the Spectrum 'explosion' is a POKE which makes the fire and debris of a bomb blast first flare up, then gradually vanish from the screen as the building collapses. This is explained more fully below.

But first you need an aircraft—and a bomb. So enter this program:

```
10 FOR n=USR "p" to USR "q"+7
20 READ a
30 POKE n,a
40 NEXT n
50 DATA 32,16,136,154,155,8,16,32
60 DATA 0,16,16,120,28,28,0,0
```

This uses conventional user defined graphics, as described on pages 38 to 45, to create images for both the aircraft and bomb. RUN the program and then check you have

entered the DATA correctly by entering (without a line number)

```
PRINT AT 1Ø,15; CHR$159; "□"; CHR$16Ø
```

which will display the two UDGs.

Next you need a building to bomb, but there is no need for a new program. Just enter the existing one, then edit it to read:

```
1Ø FOR n=USR "r" to USR "r"+7
2Ø READ a
3Ø POKE n,a
4Ø NEXT n
5Ø DATA 255,153,255,153,255,153,255,255
```

RUN the program again, and test this bit too by entering (again without a line number)

```
PRINT AT 2Ø, 15; CHR$ 161
```

Once you are satisfied, enter NEW to clear away the old program—the user defined graphics will, of course, remain in memory unless you disconnect the power.

THE BOMBING RUN

Now to the bombing program proper. Start by entering these lines:

```
1Ø BORDER Ø: PAPER 5: INK Ø: CLS
2Ø LET a$="□□□□□□□□
   □□□□□□□□"
2ØØ PRINT PAPER 4;AT 2Ø,Ø;a$;a$;
   a$;a$
21Ø PRINT INK 1;AT 19,12;CHR$ 161;
   CHR$ 161;CHR$ 161
```

What these lines do, as you'll see if you RUN them, is to PRINT a band of green, 16 pixels deep, across the bottom of the screen, with a warehouse building on top. How they do it illustrates one use of strings in graphics work, as described on page 95.

Because you need 64 coloured blocks to make up the 'grass', Line 2Ø sets up a string of 16 blank squares. Then Line 2ØØ PRINTS them four times, starting at the beginning of Line 2Ø on your screen display and spilling over into Line 21. This saves you the trouble of typing a$ 64 times!

The warehouse is built simply by PRINTing graphics character 'r' (CHR$ 161)—which you have already set up as a UDG for the building—four times on top of the grass.

To get the aircraft off the ground and into action, you need the following lines:

```
215 PAUSE 1ØØ
22Ø LET ay=6: LET by=ay
23Ø FOR x=Ø TO 3Ø
24Ø PRINT AT ay,x;"□";CHR$ 159
25Ø LET bx=x
26Ø IF by<19 THEN PRINT AT by+1,
   bx+1;CHR$ 16Ø;AT by,bx;"□"
27Ø LET by=by+1: LET bx=bx+1
28Ø IF x>29 THEN PRINT AT ay,x+1;"□"
29Ø NEXT x
```

There's nothing unusual about these: they are conventional 'moving about the screen' lines as described on pages 57 to 58 and in almost every Games Programming article since. Note, though, how the spaces in Lines 24Ø, 26Ø and 28Ø are needed to blank out the 'last positions' of both aircraft and bomb as they move on.

THE BIG BANG

Now to the main part of the exercise—the explosion itself. It is much easier to illustrate than to describe how it works. So—before you enter the rest of the program—type in these lines:

```
1ØØØ FOR n=88 to 8Ø STEP -1
1Ø1Ø PRINT AT 1Ø,15; CHR$ 15Ø
1Ø2Ø POKE 23675, n
1Ø3Ø PAUSE 5Ø
1Ø4Ø NEXT n
1Ø5Ø STOP
```

Do not just type RUN to test this program. You might corrupt the program already entered. But if you type RUN 1ØØØ, you'll see the letter G appear on the screen, then gradually vanish to be replaced by the letter F.

This is because your FOR ... NEXT loop is gradually stripping one number at a time from the value stored in memory location 23675, the starting point for your UDGs. So the letter displayed is gradually working its way back through the alphabet.

The explosion program works in much the same way. For safety's sake, enter

```
POKE 23675, 88
```

(which restores the memory location to its original value) and then delete Lines 1ØØØ to 1Ø5Ø altogether. Then you can enter the rest of the bombing program:

```
9Ø POKE 23675,88
1ØØ FOR n=Ø TO 31: READ a: POKE USR
   "a"+n,a: NEXT n
3ØØ FOR e=88 TO 8Ø STEP -1
31Ø POKE 23675,e
32Ø PRINT INK 2;AT 19,12;CHR$ 145;
   CHR$ 147;CHR$ 145: PAUSE 6
33Ø PRINT INK 2; AT 19,12;CHR$ 147;
   CHR$ 145;CHR$ 147: PAUSE 6
34Ø NEXT e
4ØØ POKE 23675,88
5ØØ GOTO 21Ø
8ØØØ DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,2,128,25,126,
   126,255,255,255
9ØØØ DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,4,33,144,66,
   231,255,255,255
```

Here, Lines 1ØØ, 8ØØØ and 9ØØØ are setting up the UDGs you need for the explosion. Line 1ØØ calls up—and POKEs into memory—the DATA in Lines 8ØØØ and 9ØØØ. Lines 3ØØ and 34Ø are carrying out much the same procedure as in your earlier experiment, stripping away one image to reveal another.

There is, however, one important difference. The UDGs creating the explosion are those based on graphics characters B and D—CHR$ 145 and 147. As they slowly disappear from the screen, you do *not* want them replaced by odd letters of the alphabet. So the first eight letters of each DATA statement, representing characters A and C—CHR$ 144 and 146—are all Øs. So, instead of getting A and C replacing B and D on the screen, what you get are blank spaces.

If you want to watch the process more closely, insert a PAUSE line between Lines 33Ø and 34Ø.

Line 4ØØ is necessary, of course, to restore the value in memory location 23675 to its original 88 before your program repeats itself.

It isn't as easy to program good visual effects on the Commodore 64 as it is on some other home computers. The most convincing explosions rely greatly on the sprite and hi-res graphic capabilities of this machine. This means that they both take quite a lengthy routine for their creation, and also usually have to be designed specifically for the program in which they are used. One example of a sprite-generated 'explosion' which could be used in several different games is given later on in this article.

Simpler routines are possible, however—such as this one for flashing bars—and can be used very effectively in programs where some sort of visual interruption is required. This one could be used to signify the end of a game or loss of a 'life':

```
1ØØ PRINT"♡"
11Ø FOR Z=Ø TO 23:READ X:POKE
   832+Z,X:NEXT Z
12Ø SYS 832:GOTO12Ø
13Ø DATA 162,Ø,16Ø,2ØØ,2ØØ,2Ø8,
   253,16Ø,25Ø,2ØØ,2Ø8,253,232
135 DATA 142,33,2Ø8,142,32,2Ø8,224,
   16,2Ø8,235,96
```

A rather more interesting routine is the one which follows, which displays a pattern of rectangles of continuously changing colours. This can be used at similar points in a games program.

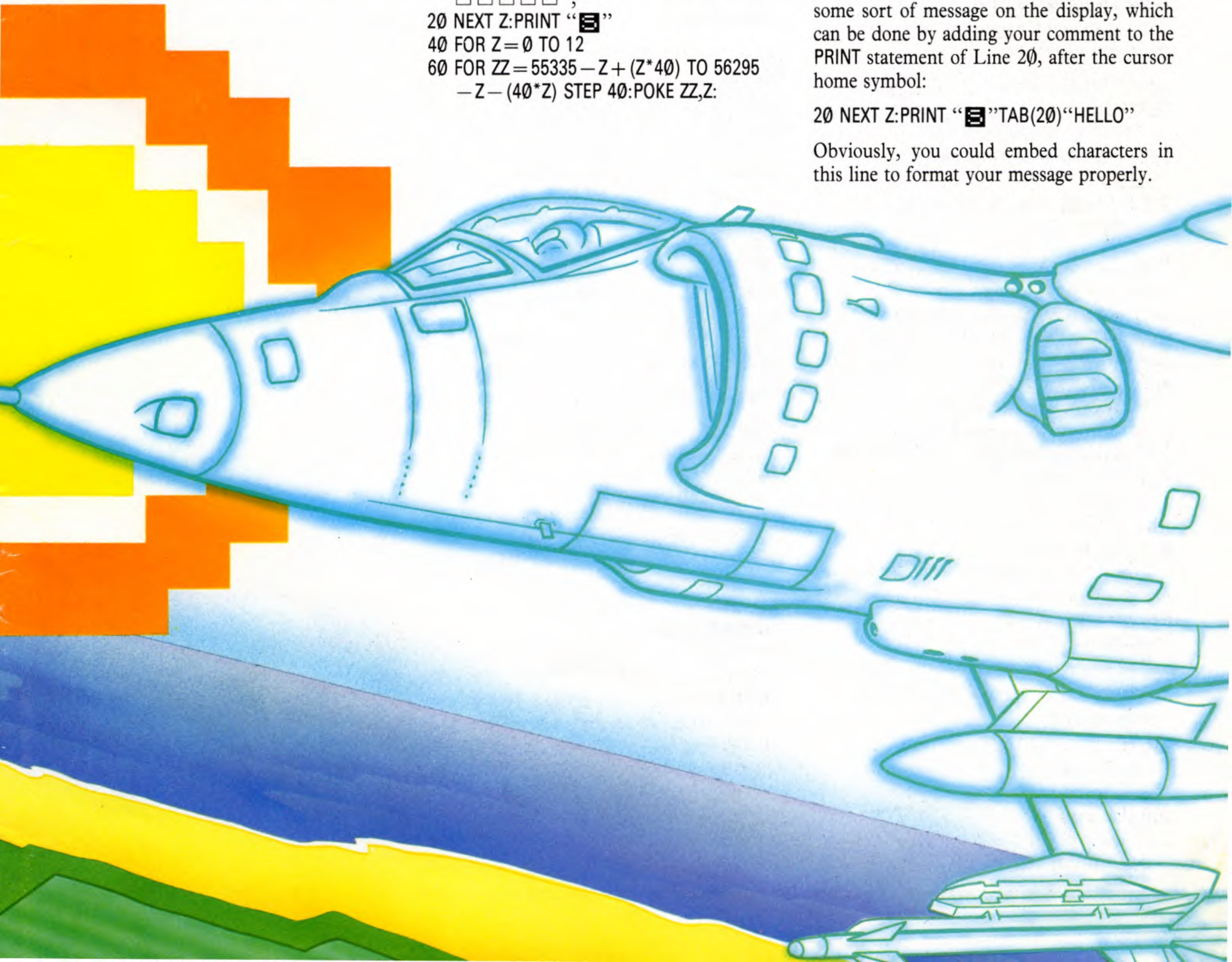
```
5 C$="■⊒╪◣▒✝⇐π♠▞
   ☒◯♣◯☒▞♠π⇐✝▒◣
   ╪⊒■":PRINT "♡";
8 FOR Z=Ø TO 48:READ X:POKE
   49152+Z,X:NEXT Z
1Ø FOR Z=1 TO 24:PRINT MID$
   (C$,Z,1)"▚□□□□□□□□□
   □□□□□□□□□□□□□
   □□□□□□□□□□□□□
   □□□□□";
2Ø NEXT Z:PRINT "▤"
4Ø FOR Z=Ø TO 12
6Ø FOR ZZ=55335-Z+(Z*4Ø) TO 56295
   -Z-(4Ø*Z) STEP 4Ø:POKE ZZ,Z:
   NEXT ZZ
7Ø FOR ZZ=55296+Z+(Z*4Ø) TO 56256
   +Z-(4Ø*Z) STEP 4Ø:POKE ZZ,Z:
   NEXT ZZ,Z
8Ø DATA 169,Ø,141,251,Ø,169,216,141,
   252,Ø,16Ø,Ø,177,251,2Ø1,Ø,2Ø8,2,169,
   16,24
82 DATA 233,Ø,145,251,24,23Ø,251,165,
   251,2Ø1,Ø,2Ø8,2,23Ø,252,165,251
84 DATA 2Ø1,232,2Ø8,224,165,252,2Ø1,
   219,2Ø8,218,96
1ØØ SYS 49152:FOR Z=1 TO 25Ø:NEXT
   Z:GOTO 1ØØ
```

The length of the routine is controlled by a FOR . . . NEXT loop in Line 1ØØ which creates a delay of 25Ø time units. This value can be adjusted as required—or try removing the loop completely for a truly eyestraining display!

By removing Lines 4Ø, 6Ø and 7Ø you lose the vertical bars, leaving horizontal bands of colour which move towards the centre. This form may be preferable if you want to include some sort of message on the display, which can be done by adding your comment to the PRINT statement of Line 2Ø, after the cursor home symbol:

```
2Ø NEXT Z:PRINT "▤"TAB(2Ø)"HELLO"
```

Obviously, you could embed characters in this line to format your message properly.



SPRITE EXPLOSION

The short program that follows is a simple demonstration of how a typical games program may look and operate if written in BASIC. This shows a jet plane flying across the screen and bombing a building.

```
10 PRINT "♥":POKE 53280,0:POKE
   53281,0
15 PRINT "▣▣▣▣▣▣▣▣
   ▣▣▣▣▣▣▣▣▣▣▣"
   TAB(20)"◘⊞◢◣▣▮▮
   π▒═□▣▮▮▮▮▮✝□□□"
20 FOR Z=1 TO 20:PRINT "☰"TAB(Z)
   "□◘▒◣⊞═▣▮▮□✝□□"
30 POKE 1024+Z*41,32:POKE 1065+Z
   *41,42:POKE 55337+Z*41,1
40 NEXT Z:PRINT "☰"TAB(Z)
   "□□▣▮▮□□□□□"
```

Without an 'explosion', the effect of the bomb isn't very realistic, so try adding these lines which make the building slowly collapse:

```
45 FOR Z=1 TO 20:POKE 1884+RND(1)
   *2,RND(1)*15+110:NEXT Z
50 POKE 1885,223:POKE 1845,32:
   POKE 1884,104
```

Even so, the resulting effect is hardly spectacular. But next add these lines, a routine for a sprite-based explosion:

```
1 FOR Z=832 TO 832+63*2:POKE Z,0:
   NEXT Z
2 FOR Z=13 TO 14:FOR A=0 TO 15:
   FOR B=0 TO 1: READ X:POKE Z*64
   +A*3+B,X:NEXT B,A,Z
3 DATA 0,0,0,0,0,0,0,15,240,15,240,
   53,92,54,156,54,156,53,92,15,240,15,240
4 DATA 0,0,0,0,0,0,0,0
5 DATA 15,240,15,240,53,92,53,92,213,
   87,214,151,214,151,218,167,218,167
6 DATA 214,151,214,151,213,87,53,92,
   53,92,15,240,15,240
900 V=53248:POKE V+21,3:POKE V+32,
   0:POKE V+33,0:POKE V+37,2:POKE
   V+28,3
910 POKE V+27,3:A(0)=1:A(1)=8:
   B(0)=4:B(1)=7
915 FOR LL=1TO3:IFLL=1ORLL=3THEN
   POKEV+23,0:POKEV+29,0:POKEV,
   183:POKEV+1,216:DD=0
920 IF LL=2 THEN POKEV+23,3:POKE
   V+29,3:POKE V,175:POKE V+1,208:
   DD=20
925 FOR UU=1 TO 10:FOR Z=13 TO
   14:POKE 2040,Z:POKE V+39,
   B(RND(1)*2)
930 FOR T=1 TO DD:NEXT T:POKE
   V+38,A(RND(1)*2)
940 NEXT Z,UU,LL:POKE V+21,0
```

RUN the program to display a colourful explosion. Such a routine can do wonderful things for even simple games programs.

Despite its apparent complexity, the program is really quite simple—basically it consists of generating two sprites and then swapping these over repeatedly (this will be explained more fully later). The DATA for the 'explosion' sprites is contained in four lines at the start of the program. Before this is the initializing routine, which first clears an area of memory (locations 832 to 1022 are actually part of the tape I/O buffer but are available for sprite storage). In the following line, DATA is READ and stored in the same, cleared, locations. Note that Z is given values of 13 and 14 which, multiplied by 64 later in Line 2, gives the start of storage of each, locations 832 and 896. The figure 64 here reflects the amount of memory each sprite requires—63+1 bytes in all.

The initializing routine must be placed at the start of any program. This is why the second part of the explosion program is numbered from Line 900, as it would typically form a subroutine within a larger program.

Line 900 *enables* the two sprites—that is, turns them on—sets the colour of screen and border to black (the value 0 in location V+32 and V+33, otherwise 53280 and 53281, may be changed) and changes the colour of the second 'ring' of each sprite to enhance the explosion effect. The last instruction on this line enables multicolour mode for both sprites.

Thereafter the program proceeds to enable the background—so you can see detail through the sprites—and colour the centre of the explosion. You could try changing the values of the numbers which follow the equal signs in Line 910, but notice how only the 'warm' colours and white prove to be effective.

The next line (Line 915) controls the pointer, which calls either the small or the large sprite, alternating the two to create the pulsing explosion effect. When values of LL are 1 and 3, the small sprite is in use and when the value is 2 the large sprite is in use. A flicker effect is added by the FOR ... NEXT loop at the start of Line 930. The explosion routine is ended in Line 940 when the sprite enable location (53269) is switched off.

The explosion routine can be added to other programs. To move the position of the sprites about adjust the values of V and V+1 in Lines 915 and 920. V is the screen X co-ordinate, V+1 is the Y co-ordinate, and the difference between the two is 33 in each case. The value of V in Line 915 is 8 more than whatever value you choose for V in Line 920. The value of V+1 in Lines 920 and 915 is then 33 more than V.

Making the screen flash is a very simple way of showing a collision in mid-air or in space. Type in this program and try RUNning it on its own:

```
7090 MODE 1
8000 FOR T=1 TO 15
8010 FOR COL=0 TO 7
8020 VDU19,0,COL;0;
8030 NEXT COL
8040 NEXT T
8050 VDU19,0,0;0;
```

What happens here is very simple. The loop in Lines 8000 and 8040 changes the screen colours 15 times—Line 8010 steps through the eight colours and then the VDU 19 redefines colour 0, the background colour, to each of the other colours in turn.

You don't actually see the whole screen change colour, however. What you actually see is bands of colour creeping up the screen. This is because the colours change so rapidly that the TV doesn't have enough time to alter the colour of the whole screen before swapping to the next colour. Only a small band of

the screen is completed each time.

If you don't like the stripey effect of the last program then add a short delay which will allow the complete screen to change colour:

```
8Ø25 FOR delay=1 TO 2ØØ : NEXT
```

Change Line 8ØØØ as well, or the flashing will last too long:

```
8ØØØ FOR T=1 TO 3
```

If you want to use the program as a subroutine in a game, then you'll have to make two alterations. First, delete Line 7Ø9Ø—as the mode will already be defined in the main program. Then add a RETURN line, such as 8Ø6Ø RETURN, to the end of the program.

If you SAVEd the alien game from Games Programming 5 then you can use that to try out the effect. Otherwise, use the simple animation program below. It shows an alien being hit by a missile:

```
1Ø MODE1
15 VDU23;82Ø2;Ø;Ø;Ø;
17 CLS
2Ø VDU23,224,6Ø,126,219,219,126,6Ø,
   92,153
3Ø VDU23,225,16,16,16,56,56,56,4Ø,1Ø8
4Ø PRINTTAB(19,4)CHR$(224)
5Ø FOR Y=31 TO 3 STEP -1
6Ø PRINTTAB(19,Y)CHR$(225)
7Ø A=INKEY(1Ø)
8Ø PRINTTAB(19,Y)"□";
9Ø NEXT
1ØØ GOSUB 8ØØØ
11Ø GOTO 17
8Ø6Ø RETURN
```

Don't forget that there must always be a line in the main program to call the subroutine. In the alien program above, this is:

```
1ØØ GOSUB 8ØØØ
```

So add a line like this (with the appropriate line numbers) whenever you want to use the subroutine in your own programs.

GROUND EXPLOSIONS

So far the program has manipulated the screen display as a whole. An alternative is to try writing some suitable animated graphics.

Here's an idea which animates some flames. They are meant to be superimposed over a target which has been successfully destroyed. You could use the flames in a game involving tanks, ships, buildings, or any ground-based target.

The program to animate the flames is written as a PROCedure so it can be added to any suitable program.

```
1Ø MODE1
2Ø VDU 5
3Ø FOR T=224 TO 229:VDU23,T
4Ø FOR P=Ø TO 7:READ A:VDU A
5Ø NEXT P
6Ø NEXT T
7Ø GCOLØ,2:MOVEØ,1ØØ:MOVE Ø,2ØØ:PLOT
   85,128Ø,1ØØ:PLOT 85,128Ø,2ØØ
8Ø Q=1:S=1:X=6ØØ:Y=232:FY=Y:
   BY=763
9Ø PROCFACTORY
1ØØ FOR T=Ø TO 12ØØ STEP 16
11Ø MOVE T,8ØØ:GCOLØ,3:VDU229
12Ø IF T>128 AND BY>25Ø THEN
    PROCBOMB
13Ø IF BY<=25Ø AND Q<16 THEN
    PROCEXPLOSION
14Ø MOVE T,8ØØ:GCOLØ,Ø:VDU226
15Ø NEXT T
16Ø GOTO 8Ø
17Ø DEF PROCBOMB
18Ø MOVE T-16,BY:GCOLØ,Ø:VDU226
19Ø BY=BY-16
2ØØ MOVE T,BY:GCOLØ,3:VDU228
21Ø ENDPROC
22Ø DEF PROCFACTORY
23Ø GCOLØ,3:MOVE 6ØØ,2ØØ:MOVE692,2ØØ:
    PLOT85,6ØØ,231:PLOT85,692,231:
    GCOLØ,2:MOVE6ØØ,263
24Ø VDU227,227,227
25Ø ENDPROC
8ØØØ DEF PROCEXPLOSION
8Ø1Ø GCOLØ,Ø:MOVE X,Y:VDU226,226,226
8Ø2Ø MOVE X,Y+32:VDU226,226,226
8Ø3Ø GCOLØ,2:MOVE X,FY
8Ø4Ø VDU224+S,224+S,224+S
8Ø5Ø S=1-S
8Ø6Ø FY=FY-2
8Ø7Ø Q=Q+1
8Ø8Ø ENDPROC
9ØØØ DATA 2,128,25,126,126,255,255,255,
     4,33,144,66,231,255,255,255
9Ø1Ø DATA 255,255,255,255,255,255,255,
     255
9Ø2Ø DATA Ø,Ø,Ø,Ø,3,15,63,255,Ø,16,16,
     12Ø,28,28,Ø,Ø
9Ø3Ø DATA 32,16,136,254,255,8,16,32
```

This program shows a plane dropping a bomb on a factory. If the factory is destroyed, then two sets of flames are PRINTed, one after the other to give the impression of a flickering fire. The flames also gradually die down as the program proceeds.

Here's how the program works, line-by-line to create the effect.

The DATA for the two sets of flames is in Line 9ØØØ and the other DATA lines are for a blank square, the plane and the bomb. Lines 3Ø to 6Ø READ the DATA and set up the UDGs, and the flames are PRINTed, at the correct position, by Lines 8Ø3Ø and 8Ø4Ø. Lines 8Ø1Ø and 8Ø2Ø simply blank out the factory before the flames are displayed. Which set of flames is PRINTed by Line 8Ø4Ø depends on the value of S. S starts off equal to 1, set in Line 8Ø, so the first time the PROCedure is called it PRINTs character 244+1, that is 245. Line 8Ø5Ø changes the value of S to Ø so the next time round, character 244+Ø, i.e. 244, is PRINTed. The third time round, S is changed back to 1 again and so on. As the characters are swapped back and forth the flames flicker and appear to be animated. Line 8Ø6Ø alters the Y co-ordinate of the flames making them move down the screen each time giving the impression that they are dying down.

Finally, Line 8Ø7Ø uses Q to keep count of how many times the flames are PRINTed and when Q=16 the condition in Line 13Ø becomes false so the procedure is not called again until the next bomb drops.

The rest of the program controls the movement of the plane and the bomb and draws the screen display. Line 7Ø PLOTs the ground at the bottom of the screen, Line 9Ø DRAWs the factory, and Lines 1ØØ, 11Ø and 14Ø move the plane across the top of the screen. PROCBOMB is called in Line 12Ø but only if the plane has reached its target at position 25Ø. When the bomb does reach its target then PROCEXPLOSION is called instead by Line 13Ø.

Dramatic visual effects that will greatly enhance space games can be achieved on the Dragon or Tandy with very short programs. Try typing in this one and RUNning it:

```
798Ø PMODE 3,1
799Ø PCLS
8ØØØ FOR F=1 TO 1ØØØ
8Ø1Ø SCREEN 1,Ø
8Ø2Ø SCREEN 1,1
8Ø3Ø NEXT F
8Ø4Ø CLS
```

You'll see bands of colour creeping down the screen, caused by the computer switching very quickly between colour sets. It's an effect you could use for the end of a particular phase of a game—penetration of a shield, perhaps?

The mode is set up by Line 798Ø and then the graphics screen is cleared by Line 799Ø. Line 8Ø1Ø switches on one colour set, whilst Line 8Ø2Ø switches back to the other. The FOR ... NEXT loop in Lines 8ØØØ and 8Ø3Ø switches the two colour sets 1000 times.

This switching changes the colour sets very rapidly—so quickly, in fact, that the TV never has enough time to complete changing the colour of the screen before it has to swap it back again. Only a small band of the screen is thus completed in each colour.

If you want to use the program as a subroutine—either with a game (it'll only work with games written in one of the PMODEs, such as that on page 144) or with the alien program given below—you'll have to make a few alterations. First, delete Lines 798Ø and 799Ø, then add a RETURN line such as 85ØØ RETURN.

```
1Ø PMODE 1,1
2Ø DIM A(3),B(3),M(3)
2ØØ FOR K=1536 TO 2Ø16 STEP 32
21Ø READ A,B:POKE K,A:POKE K+1,B
22Ø NEXT
23Ø GET(Ø,Ø)-(15,15),A,G
24Ø GET(Ø,16)-(15,31),M,G
25Ø PCLS
26Ø MX=12Ø:MY=191:PX=12Ø:PY=2Ø
27Ø PUT(PX,PY)-(PX+15,PY+15),
   A,PSET
28Ø SCREEN 1,Ø
29Ø PUT (MX,MY)-(MX+15,MY+15),
   B,PSET
3ØØ MY=MY-4
31Ø IF MY<36 THEN GOSUB 8ØØØ:
   GOTO 26Ø
32Ø PUT (MX,MY)-(MX+15,MY+15),
   M,PSET
33Ø GOTO 29Ø
9ØØØ DATA 252,63,3,192,15,24Ø,61,124,
   58,172,245,95,213,87,213,87
9Ø1Ø DATA Ø,128,Ø,128,Ø,128,2,16Ø,1Ø,
   168,Ø,192,3,24Ø,15,6Ø
```

The program shows an alien being hit by a missile. When the alien is hit, Line 31Ø calls up the screen stripe routine.

When used as a subroutine, the screen stripe program will make stripes in any PMODE you like, so it can be added to any game without altering Lines 8ØØØ to 8Ø4Ø.

Here's an adaptation of the above stripes program which gives a more elaborate effect, although the screen graphics won't remain intact as they did in the first case. You can type it in and RUN it as it is, or add it to another program—such as the alien animation—as a subroutine. If you use it as a subroutine, you'll need to delete Line 799Ø and add a RETURN line—as you did before.

```
799Ø PMODE 3,1
8ØØØ FOR F=1 TO 3
8Ø1Ø FOR K=Ø TO 1
8Ø2Ø SCREEN 1,K
8Ø3Ø FOR J=1 TO 4
8Ø4Ø PCLS J
8Ø5Ø NEXT J
8Ø6Ø NEXT K
8Ø7Ø NEXT F
8Ø8Ø CLS
```

As a subroutine it will work in both the two-colour modes (PMODEs Ø, 2 and 4), and the four-colour modes (PMODEs 1 and 3). In addition to changing the colour sets, the routine also clears the screen and changes the screen colour.

Lines 8Ø3Ø to 8Ø5Ø are a FOR ... NEXT loop which turns the screen to each of the available colours in the colour set in turn. The program is designed to look for four colours in the set, but will work in a two-colour mode despite this. No errors will be reported and only two colours will appear on the screen.

One last variation on the program looks like this:

```
799Ø PMODE 3,1
8ØØØ FOR F=1 TO 5
8Ø1Ø SCREEN 1,Ø
8Ø2Ø FOR K=1 TO 2ØØ: NEXT K
8Ø3Ø SCREEN 1,1
8Ø4Ø FOR K=1 TO 2ØØ: NEXT K
8Ø5Ø NEXT F
8Ø6Ø CLS
```

Either type it in as it is, or use it as a subroutine by deleting Line 799Ø and adding a RETURN line at the end.

You will see the screen flashing in colour. This happens because Lines 8Ø2Ø and 8Ø4Ø insert short delays allowing the complete screen to change colour before the colour sets are swapped. You can experiment by altering the length of the delay and the number of times that the colours are swapped.

FLAMES

So far you've seen some ideas for manipulating the screen display as a whole. But even better visual effects can be achieved by writing some suitable animated graphics.

Here's an idea which animates some flames

that you could superimpose over a target which has been successfully destroyed. Alternatively, you could use them in a game involving any ground-based target.

The program has been written in PMODE 1, and can't be used as it stands with games written in other PMODEs. Type it in and RUN it to see what it looks like:

```
10 PMODE 1,1
2Ø DIM B(3),E1(3),E2(3)
3Ø FOR K=1536 TO 2Ø16 STEP 32
4Ø READ A,B:POKE K,A:POKE K+1,B
5Ø NEXT
6Ø GET (Ø,Ø)-(15,15),E1,G
7Ø GET (Ø,16)-(15,31),E2,G
8Ø SCREEN 1,Ø
25Ø PCLS:HX=124:HY=146
8ØØØ FOR N=Ø TO 15
8Ø1Ø PUT (HX,HY+N)-(HX+15,HY+15),
   E1,PSET
8Ø2Ø FOR K=1 TO 1ØØ:NEXT
8Ø3Ø PUT (HX,HY+N)-(HX+15,HY+15),
   E2,PSET
8Ø4Ø FOR K=1 TO 1ØØ:NEXT
8Ø5Ø PUT (HX,HY+N)-(HX+15,HY+15),
   B,PSET
8Ø6Ø NEXT
9ØØØ DATA Ø,12,192,Ø,3,195,63,252,
   63,252,255,255,255,255,255,255
9Ø1Ø DATA Ø,48,12,3,195,Ø,48,12,252,
   63,255,255,255,255,255,255
```

Two sets of flames are PUT on the screen, one after the other, giving the impression of fire. The flames also gradually die down.

The program works like this:

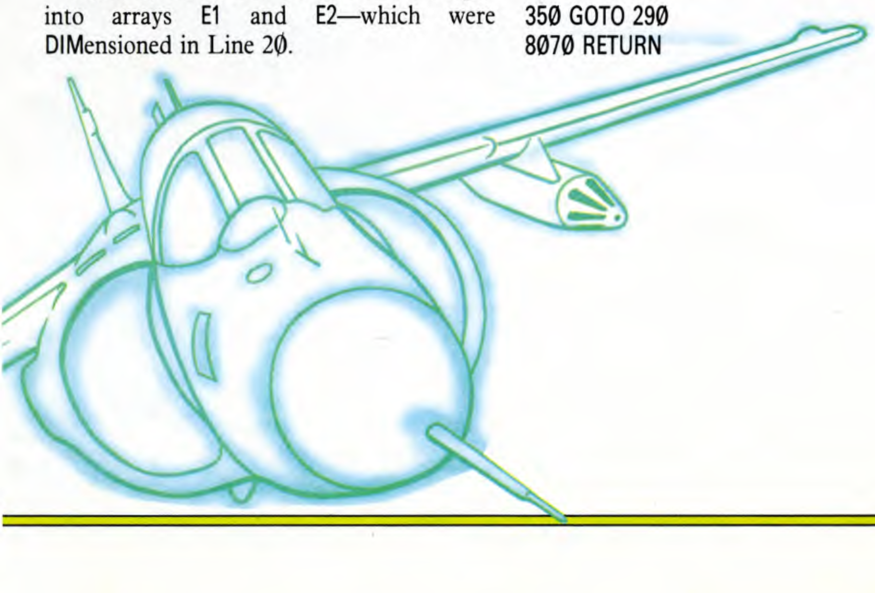
The DATA for the flames is in Lines 9ØØØ and 9Ø1Ø. The flames are POKEd on to the screen by Lines 3Ø to 5Ø. Because the program is written in a four-colour mode the DATA is a little different from the DATA you saw in Machine Code 2 (page 40). Setting up colour UDGs will be dealt with in detail later.

Lines 6Ø to 7Ø GET the two flame shapes into arrays E1 and E2—which were DIMensioned in Line 2Ø.

Lines 8ØØØ to 8Ø6Ø animate the flames and make them die down. Each time the program goes through the FOR ... NEXT loop the two flame arrays are PUT on the screen, and blanked out by PUTting the blank array, B, on top of them. The top of the flames is lowered each time by the +N in the PUT lines, so the flames appear to be dying down. If you don't see how this works try plotting the coordinates in the PUT lines as the program goes through the FOR ... NEXT loop.

Although the program will work on its own, it'll look a lot better if you try it out as a subroutine with the bombing run program given below—you'll see a building being destroyed and then burned down. If you use the two programs together make sure that Line 2Ø is altered so that it appears in the bombing run—use the machine's editor, or retype the line. Also, delete Line 8Ø—just type 8Ø ENTER.

```
2Ø DIM A(3),B(3),H(3),E1(3),E2(3)
2ØØ FOR K=1536 TO 2Ø16 STEP 32
21Ø READ A,B:POKE K,A:POKE K+1,B
22Ø NEXT
23Ø GET(Ø,Ø)-(15,15),A,G
24Ø GET(Ø,16)-(15,31),H,G
25Ø PCLS:LINE(Ø,163)-(255,191),
   PSET,BF
26Ø HX=124:HY=146:PX=Ø:PY=4Ø:B=Ø
27Ø PUT(HX,HY)-(HX+15,HY+15),H,PSET
28Ø SCREEN 1,Ø
29Ø PUT(PX,PY)-(PX+15,PY+15),B, PSET
3ØØ PX=PX+4
31Ø PUT(PX,PY)-(PX+15,PY+15),A,
   PSET
32Ø IF PX=2Ø THEN B=1:BX=PX+8:
   BY=PY+8
33Ø IF B=1 THEN PRESET(BX,BY):
   PRESET(BX+2,BY):BX=BX+2:BY=
   BY+2:PSET(BX,BY,4):PSET(BX+2,BY,4)
34Ø IF BY=148 THEN GOSUB 8ØØØ: BY=Ø:
   GOTO 25Ø
35Ø GOTO 29Ø
8Ø7Ø RETURN
```



```
9Ø2Ø DATA Ø,Ø,2,Ø,13Ø,128,162,16Ø,17Ø,
   17Ø,162,16Ø,13Ø,128,2,Ø
9Ø3Ø DATA Ø,3,12,51,6Ø,243,255,255,
   255,255,85,85,86,149,86,149
```

If you want to use the subroutine in other PMODEs it will have to be changed slightly. Here are two versions which will allow the flames to be used with games written in other modes.

First, here is a program suitable for use in PMODEs 3 and 4—change the mode number as appropriate:

```
1Ø PMODE 3,1
2Ø DIM B(6),E1(6),E2(6)
3Ø FOR K=1536 TO 2496 STEP 64
4Ø READ A,B:POKE K,A:POKE K+1,B
45 POKE K+32,A:POKE K+33,B
5Ø NEXT
6Ø GET(Ø,Ø)-(15,15),E1,G
7Ø GET(Ø,16)-(15,31),E2,G
8Ø SCREEN 1,Ø
25Ø PCLS: HX=124:HY=146
8ØØØ FOR N=Ø TO 15
8Ø1Ø PUT(HX,HY+N)-(HX+15,HY+15),
   E1,PSET
8Ø2Ø FOR K=1 TO 1ØØ:NEXT
8Ø3Ø PUT(HX,HY+N)-(HX+15,HY+15),
   E2,PSET
8Ø4Ø FOR K=1 TO 1ØØ:NEXT
8Ø5Ø PUT(HX,HY+N)-(HX+15,HY+15),
   B,PSET
8Ø6Ø NEXT
9ØØØ DATA Ø,12,192,Ø,3,195,63,252,63,
   252,255,255,255,255,255,255
9Ø1Ø DATA Ø,48,12,3,195,Ø,48,12,252,63,
   255,255,255,255,255,255
```

And secondly, here is a program suitable for use in PMODE 2.

```
1Ø PMODE 2,1
2Ø DIM B(6),E1(6),E2(6)
3Ø FOR K=1536 TO 2496 STEP 32
4Ø READ A:POKE K,A:POKE K+16,A
5Ø NEXT
6Ø GET(Ø,Ø)-(15,15),E1,G
7Ø GET(Ø,16)-(15,31),E2,G
8Ø SCREEN 1,Ø
25Ø PCLS: HX=124:HY=146
8ØØØ FOR N=Ø TO 15
8Ø1Ø PUT(HX,HY+N)-(HX+15,HY+15),
   E1,PSET
8Ø2Ø FOR K=1 TO 1ØØ:NEXT
8Ø3Ø PUT(HX,HY+N)-(HX+15,HY+15),
   E2,PSET
8Ø4Ø FOR K=1 TO 1ØØ:NEXT
8Ø5Ø PUT(HX,HY+N)-(HX+15,HY+15),
   B,PSET
8Ø6Ø NEXT
9ØØØ DATA 2,128,25,126,126,255,255,255
9Ø1Ø DATA 4,33,144,66,231,255,255,255
```

SPRITES ON THE COMMODORE 64

The sprite is a standard graphics facility on the Commodore 64, producing a very mobile high resolution image. Simple to master, it's the basis of many games.

A sprite—also called a movable object block (MOB)—is a special kind of highly mobile high resolution UDG some uses of which have already been seen. Apart from its mobility, it has other unusual characteristics—for example, the ability to expand and contract in width and height upon command. It is not therefore surprising to find that sprites are very commonly used in games programming—but, in fact, sprites can be used in any program where high resolution animated graphics are required. A business program, for example, could make use of one or more sprites to form an 'icon' (symbol depicting choice of program operation), or as part of an animated logo.

There are two types of sprite: standard high resolution form or multicolour. The difference between the two is that the former adopt the chosen foreground colour. Multicolour sprites offer a choice of up to four colours simultaneously, but only by sacrificing some of the horizontal resolution.

To start with, let's look at standard sprites. Multicolour sprites are covered separately.

DEFINING A SPRITE

Sprites are defined very much like standard UDGs (see page 38). But a sprite is larger and so needs more DATA to define it. In fact, it is three times the width of a standard UDG, and not quite three times its depth, occupying an area 24 pixels by 21. Not all of the 504 available pixel positions have to be used (that is, turned on)—and it may not be possible to do so in certain instances.

Rather than defining each of the 24 pixels on each row separately, information about the sprite shape is contained in three groups of eight (three bytes) on each of the 21 rows. This gives a key to the way DATA statement values used for normal UDGs can be adapted for use in this case, since these are also calculated in groups of eight.

The arrangement of bytes in the sprite takes the following form:

Row 1:	BYTE 1	BYTE 2	BYTE 3
Row 2:	BYTE 4	BYTE 5	BYTE 6
Row 3:	BYTE 7	BYTE 8	BYTE 9

. . . and continues up to:

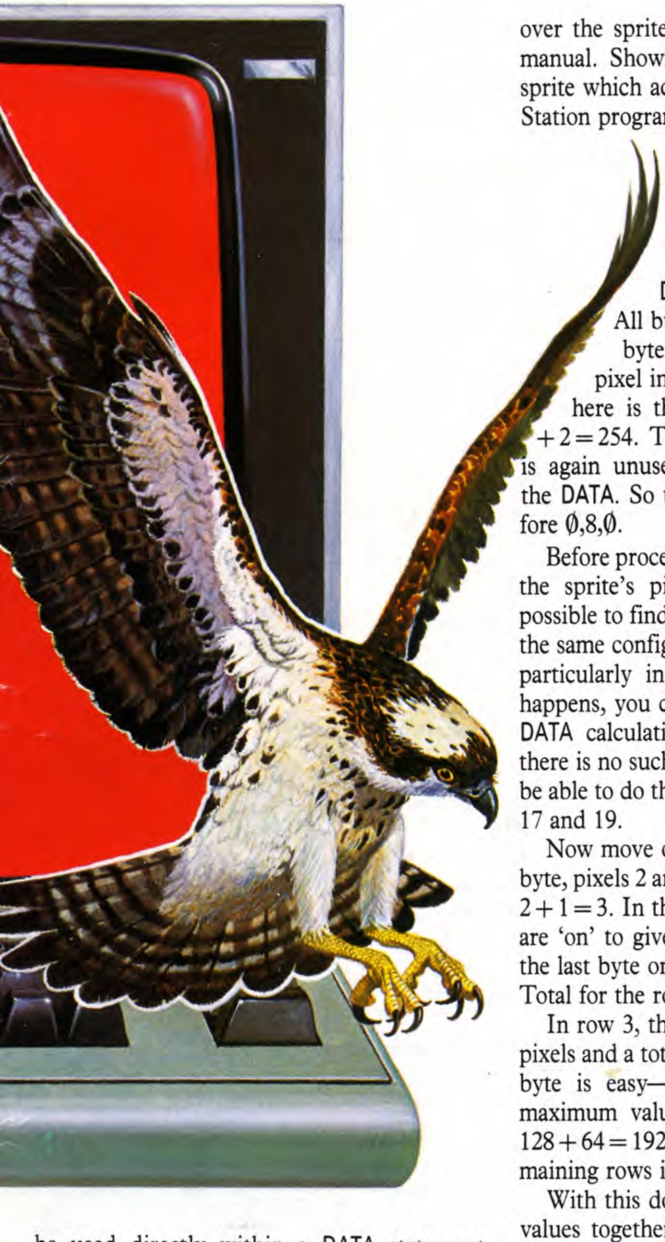
Row 2Ø:	BYTE 58	BYTE 59	BYTE 6Ø
Row 21:	BYTE 61	BYTE 62	BYTE 63

So you have side by side groupings of three bytes for each of the 21 sprite rows.

Each of these bytes handles information in the same way as a single line of the UDG on page 38. If you think in terms of decimal notation, for instance, the possible values for the pixel positions across each row are shown in the diagram on page 17Ø.

So if all the eight available pixel positions (bits) in any one byte are used (that is, switched on or *enabled*) the corresponding decimal value of that byte is 128+64+32+16+8+4+2+1, which is 255. If no pixels are used, the sum of the bit values is Ø. Between the two extremes are unique values corresponding to any possible permutation of 'used' pixels. The value obtained can

- WHAT IS A SPRITE?
- DEFINING A SPRITE
- FIRST STEPS
- **DATA** GENERATION
- USING THE SPRITE
- SPRITES IN MEMORY
- A NEAT TRICK FOR TRACING OFF YOUR DESIGN
- AVOIDING **SAVE** AND **LOAD** PROBLEMS



be used directly within a DATA statement as part of a sprite definition.

FIRST STEPS

Now let's look at some examples. As with ordinary UDGs, the best way to start is by designing the required shape on suitably ruled graph paper. Mark out an area containing 24 by 21 squares, and indicate the byte divisions. Alternatively, photocopy or trace over the sprite grid provided here or in the manual. Shown overleaf is an example of a sprite which actually forms part of the Space Station program on page 151.

This example shows the DATA values alongside. Let's take a closer look at these. The first byte in the first row is not used—so the DATA value of these pixels is Ø. All but one of the pixels in the next byte are activated, with the unused pixel in the last place. The total value here is thus 128+64+32+16+8+4+2=254. The third byte in the first row is again unused, contributing another Ø to the DATA. So the values for row 1 are therefore Ø,8,Ø.

Before proceeding, have a look at the rest of the sprite's pixel arrangement. It is often possible to find another row which uses much the same configuration for one or more bytes, particularly in symmetrical designs. If this happens, you can save having to perform the DATA calculation more than once. In fact, there is no such match for row 1, but you will be able to do this with rows 14 and 16 or rows 17 and 19.

Now move on to the next row. In the first byte, pixels 2 and 1 are 'on', yielding a value of 2+1=3. In the next byte, pixels 6,5,4 and 1 are 'on' to give 32+16+8+1=57. And in the last byte only the first pixel (128) is used. Total for the row is thus 3,57,128.

In row 3, the first byte has three activated pixels and a total of 4+2+1=7. The second byte is easy—all the pixels are on, for a maximum value of 255. The last byte has 128+64=192. Check the values for the remaining rows in the same way.

With this done, you can group all the row values together to obtain a DATA statement. For the space station sprite this is:

```
2Ø DATA Ø,254,Ø,3,57,128,7,255,192,Ø,
   16,Ø,16,56,16,56,84,56,124,148,124,
   131,255
3Ø DATA 13Ø,144,58,18,184,16,58,144,16,
   18,131,255,13Ø,254,84,254,252,56,126,
   Ø,56
4Ø DATA Ø,Ø,4Ø,Ø,Ø,56,Ø,1,199,Ø,6,16,
   192,1,199,Ø,Ø,124,Ø
```

This is how this sprite appears in the Space Station program on page 151. But it isn't always necessary to write it in full this way. With suitable program adjustments so that these values aren't expected, it is usually possible to omit unused sprite rows at the start and end of a definition.

The remaining examples are less complicated designs, and calculating the values is much more straightforward. See if you can follow how the values are obtained, and whether you can find them in the DATA statements which are located in Lines 2Ø to 14Ø of the Space Station program.

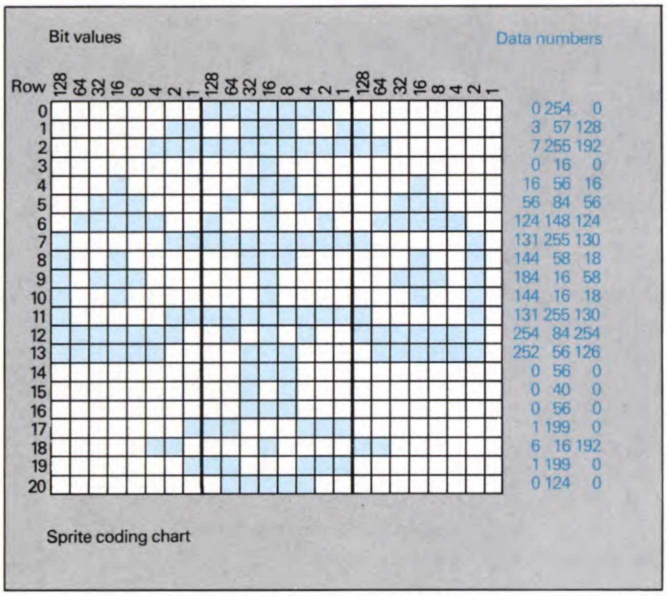
DATA GENERATION PROGRAM

The least exciting part of working with a sprite is calculating the 63 DATA statement values needed to define it. Instead of doing this by hand, you can use a utility program to work out the values for you, and a wide choice of products of this type is available commercially.

But all you need to start with is a comparatively simple program which lets you 'draw' the sprite design on the screen, altering it as you wish. It will then automatically calculate the corresponding DATA values which you can use to define the sprite for use in other programs.

The program which follows does just this, and gives you the option, if you have a printer, of taking a hardcopy for future reference. The program is suitable only for single-colour high resolution sprites—but is nonetheless extremely useful. Although typing it in may seem like a long-winded way of doing one set of sprite calculations, when you have done it once you can SAVE the program for re-use every time you have a sprite to define:

```
1Ø POKE 5328Ø,1:POKE 53281,1:DIM
   A$(21),Z(3,21),A(24)
2Ø PRINT "♥▓"TAB(13)"SPRITE
   EDITOR▟▟"
3Ø PRINT TAB(8);:INPUT "▬PRINTER
   OPTION.(✝Y/N▬)✝";I$:IF I$="Y"
   THEN PR$="Y"
4Ø IF I$<>"Y" AND I$<>"N" THEN
   GOTO 2Ø
5Ø PRINT"♥╧"TAB(13)"PLEASE WAIT"
```



Row	Data numbers		
0	0	254	0
1	3	57	128
2	7	255	192
3	0	16	0
4	16	56	16
5	56	84	56
6	124	148	124
7	131	255	130
8	144	58	18
9	184	16	58
10	144	16	18
11	131	255	130
12	254	84	254
13	252	56	126
14	0	56	0
15	0	40	0
16	0	56	0
17	1	199	0
18	6	16	192
19	1	199	0
20	0	124	0

An example of a sprite from the Space Station game—the station itself—shown superimposed on a grid giving row and bit numbers. See how the bit values can be added to form the DATA numbers. There are three numbers for each row because each one is made up of three bytes containing eight pixels each (see text). If you like, you can trace over this grid and use it when you are designing your own sprites and wish to work out the DATA

```
60 FOR Z=1 TO 8:READA(Z):A(Z+8)=
   A(Z):A(16+Z)=A(Z):NEXT:DATA128,64,
   32,16,8,4,2,1
70 FOR Z=1 TO 21:READ A$(Z)
80 FOR ZZ=1 TO 8:IF MID$(A$(Z),
   ZZ,1)=“*”THEN Z(1,Z)=Z(1,Z)+
   A(ZZ)
90 NEXT ZZ:FOR ZZ=9 TO 16:IF
   MID$(A$(Z),ZZ,1)=“*” THEN
   Z(2,Z)=Z(2,Z)+A(ZZ)
100 NEXT ZZ:FOR ZZ=17 TO 24:IF
   MID$(A$(Z),ZZ,1)=“*” THEN
   Z(3,Z)=Z(3,Z)+A(ZZ)
110 NEXT ZZ,Z:PRINT “♡■”:IF
   PR$=“Y” THEN OPEN4,4:CMD4
120 PRINT TAB(12)“■CHARACTER
   DATA■■”
130 FOR Z=1 TO 21:PRINT Z(1,Z);“,”;
   Z(2,Z);“,”;Z(3,Z);:IF Z<21 THEN
   PRINT“,”;
140 NEXT Z:PRINT:PRINT:IF PR$=“Y”
   THEN GOTO 170
150 PRINT TAB(7)“■▒ PRESS RETURN
   TO CONTINUE”
160 GET K$:IF K$< >CHR$(13) THEN
   GOTO 160
170 PRINT “♡”TAB(11)“■CHARACTER
   DESIGN■■”
180 PRINT TAB(7)“▒765432107654321Ø
   7654321Ø”
190 FOR Z=1 TO 21:PRINT TAB(7)“■”;
200 FOR ZZ=1 TO 24:IF MID$(A$(Z),
   ZZ,1)=“*” THEN PRINT “■□■”;:
   GOTO 220
210 PRINT “.”;
220 NEXT ZZ:PRINT“▒”;Z:NEXT Z:IF PR$
   =“Y” THEN PRINT #4, “□”:
   CLOSE4
230 IF PR$< >“Y” THEN GOTO 230
240 REM□□765432107654321Ø7654321Ø
250 DATA“
260 DATA“
270 DATA“
280 DATA“
290 DATA“
300 DATA“
310 DATA“
320 DATA“
330 DATA“
340 DATA“
350 DATA“
360 DATA“
370 DATA“
380 DATA“
390 DATA“
400 DATA“
410 DATA“
420 DATA“
430 DATA“
440 DATA“
450 DATA“
```

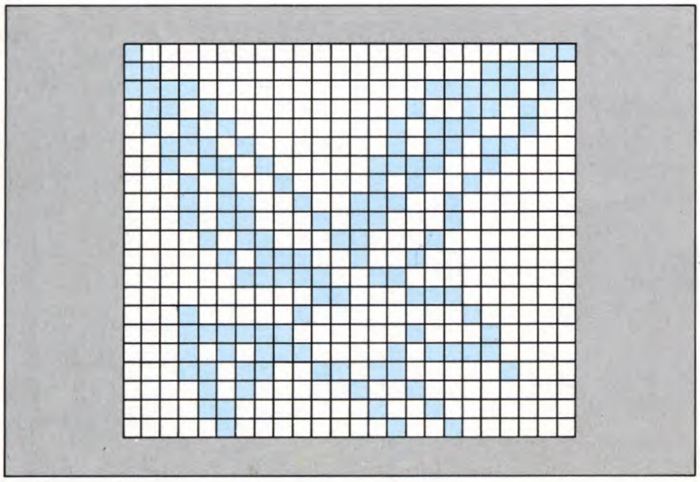
First design your flying bird on graph paper. It can then be entered as asterisks in the data generator program

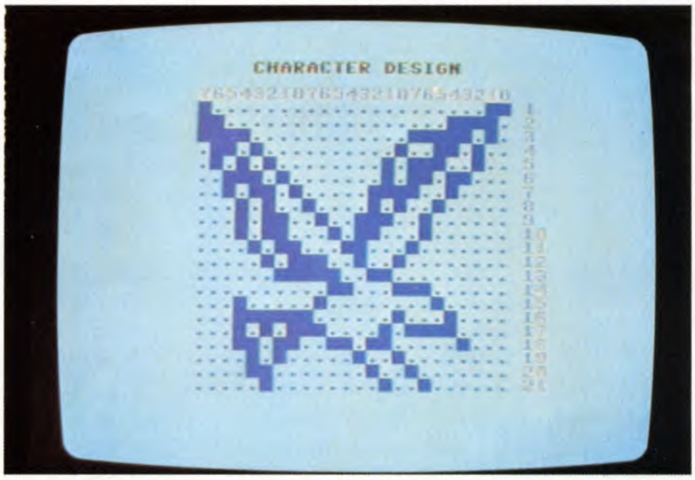
To use the program, enter the instruction LIST 240–. This will display the 21 rows at the end of the program—empty DATA lines on which you can plot the shape of the sprite you want to design. Each character position within the DATA statement represents an available pixel position—the line is 24 characters wide. Use the cursor controls to move around the lines, then simply place an asterisk at every point of your design and remember to press RETURN to enter each line.

For guidance, the REM statement in Line 240 numbers each of the 24 horizontal positions which are available for each of the 21 vertical rows.

A typical design is shown bottom left. This shows the pattern of asterisks used to define the figure of a flying bird.

When you have completed your design, RUN the program. The screen then displays a prompt asking you whether direct printout is required. If you press N, output is to the





```
CHARACTER DESIGN
765432107654321076543210
```

screen. First come the 63 DATA statement numbers—these you'll have to copy out by hand if you do not have a printer. For the example shown, the DATA looks like this:

```
CHARACTER DATA 128,Ø,3,192,Ø,3Ø,24Ø,
   Ø,25Ø,1Ø4,1,52,84,2,228,58,2,216,45,
   7,144,2Ø,133,32,22,78,64,22,44,64,11,
   24,128,5,2Ø1,Ø,3,23Ø,Ø,Ø,49,192,16,64,
   32,31,129,16,21,194,24Ø,27,54,8,1Ø,9,Ø,
   12,4,128,4,2,64
```

A large scale display of your sprite can then be obtained simply by pressing RETURN.

If you now choose to take a hardcopy using the printer, simultaneously press RUN/STOP and RESTORE and reRUN the program. Press Y when the printer option again shows. The printer output starts with a listing of all the character DATA, and this is followed by a large scale printout of the sprite on its entry grid.

Each sprite you define within this program may be SAVEd as part of it. You can then reLOAD it at will and change it as necessary to create further sprites.

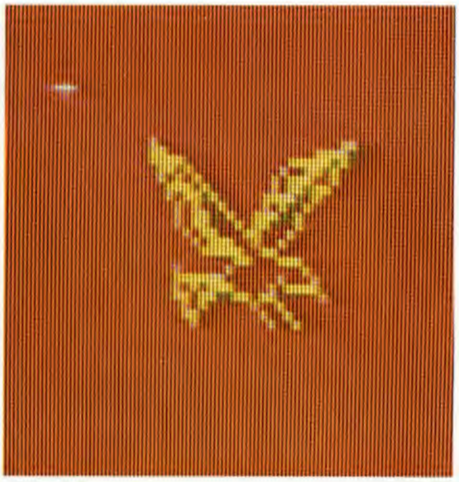
USING THE SPRITE

The design and DATA definition stage is perhaps the easiest part of using a sprite. Whichever way you work out your DATA values, they are of no use by themselves. It's only when you incorporate them in a new program that you can make use of the sprite which you have created.

First you need to store the sprite information somewhere suitable in the computer's memory. Then, you can call it up and manipulate it with your program.

Exactly how this is done is covered a little later. But for the moment, type in and RUN this program:

```
10 V=53248:X=15Ø:Y=157:PRINT "♥"
2Ø FOR I=16ØØØ TO 16Ø62:READ A:POKE
   I,A:NEXT I
25 POKE 2Ø4Ø,25Ø:POKE V+21,1:GOTO
   5Ø
3Ø GET A$:A=Ø:XX=Ø:IF A$="P" THEN
   A=1:GOTO 5Ø
35 IF A$="L" THEN A=2:GOTO 5Ø
4Ø IF A$="Z" THEN XX=-2
45 IF A$="X" THEN XX=+2
5Ø FOR Z=1 TO 1Ø:X=X+XX:IF X>25Ø
   THEN X=3Ø
55 IF X<2Ø THEN X=25Ø
6Ø IF A=1 AND Y>7Ø THEN Y=Y-2
65 IF A=2 AND Y<2ØØ THEN Y=Y+2
7Ø POKE V,X:POKE V+1,Y
75 NEXT Z:GOTO 3Ø
1ØØ DATA 128,Ø,3,192,Ø,3Ø,24Ø,Ø,
   25Ø,1Ø4,1,52,84,2,228,58,2,216,45,
   7,144,2Ø
1Ø5 DATA 133,32,22,78,64,22,44,
   64,11,24,128,5,2Ø1,Ø,3,23Ø,Ø,Ø,49,
   192,16,64
11Ø DATA 32,31,129,16,21,194,24Ø,
   27,54,8,1Ø,9,Ø,12,4,128,4,2,64
```



This instructs the computer to set up a sprite and then move the image across the screen under keyboard control using the L, P, Z and X keys. In this case the sprite chosen is the flying bird figure defined earlier, but you can use the same program for any other single-colour sprite you invent—all you have to amend is the DATA statements at the end.

If you look at lines 1ØØ, 1Ø5 and 11Ø, you'll see that these list the DATA values for the bird which were generated by the utility program.

Line 1Ø sets up a series of variables—basically a starting point for the sprite—and clears the screen. Line 2Ø READs the DATA which defines the sprite and holds it in the computer's memory. Line 3Ø then *enables* or turns the image on.

Lines 5Ø to 75 instruct the computer how to move the sprite under keyboard control. If you just want to display the sprite, try editing out Lines 4Ø, 5Ø, 55, 6Ø, 7Ø, and 75 from the program. In this case the bird will appear at a point fixed by the variables X and Y—which are set at 15Ø and 157, somewhere near the middle of the screen. If you enter alternative values for X and Y in Line 1Ø, you can make the bird appear wherever you like.

Microtip

If you're using the sprite numbers/display program here, there's a quick way of transfering a design to the DATA statement lines of the program. Using tracing paper, mark off the key lines of your design—which should match the screen in size—and then tape this on the TV screen once you have LISTed Lines 24Ø–45Ø. Now simply use the cursor and plot asterisks 'underneath' the lines of your tracing until the arrangement of these closely matches the design on the tracing paper.

The most complicated part of what's going on here is in Lines 1Ø to 25—the part that actually stores the sprite in the computer's memory. This is looked at in detail below. But don't worry if you don't understand it completely for the moment. How memory works is covered in depth in a later article.

SPRITES IN MEMORY

A sprite definition requires 63 bytes of memory for the 63 DATA values. But it is convenient to allocate 64 bytes for each, as this simplifies some of the calculations. These 64

Add the extra program line to the text program for a simple bird-zapping game using your newly defined sprite

```
15 POKE 5328Ø,2:POKE 53281,2:POKE 65Ø,
   128:FOR Z=16ØØØ TO 16ØØØ+64*2:
   POKEZ,Ø:NEXT
2Ø FOR I=16ØØØ TO 16Ø77:READ A:
   POKE I,A:NEXT I:TI$="ØØØØØØ":
   POKE V+29,1
25 POKE 2Ø4Ø,25Ø:POKE 2Ø41,251:
   POKE V+21,3:XX=1ØØ:YY=1ØØ:
   POKE V+4Ø,1
3Ø FOR Z=1 TO 5:PRINT "▤◢TIME:
   ▟"VAL(TI$):IF RND(1)>.3Ø THEN
   POKEV+23,RND(1)*2
35 A=INT(RND(1)*3)+1:X=X+1Ø:
   IF X>239 THEN X=3Ø
4Ø IF A=1 AND Y>7Ø THEN Y=Y-1Ø
45 IF A=2 AND Y<2ØØ THEN Y=Y+1Ø
5Ø POKE V,X:POKE V+1,Y:POKE V+39,7:
   GET Z$:IF Z$="Z" AND XX>3Ø THEN
   XX=XX-5
55 IF Z$="X" AND XX<25Ø THEN
   XX=XX+5
6Ø IF Z$="P" AND YY>5Ø THEN
   YY=YY-5
65 IF Z$="L" AND YY<22Ø THEN
   YY=YY+5
7Ø POKE V+2,XX:POKE V+3,YY:
   IF PEEK(V+3Ø)=3 THEN S=S+1
75 PRINT "▤◢"TAB(25)"SCORE:
   ▟"S:NEXT Z:IF VAL(TI$)<59 THEN 3Ø
8Ø PRINT "▤■▩TIMES UP ◢!":
   END
115 DATA 8,Ø,Ø,8,Ø,Ø,62,Ø,Ø,8,Ø,Ø,8,Ø,Ø
```

bytes can be stored in any area of spare memory providing it is a multiple of 64.

Each of these definitions has to be located once it has been stored, and to do this *sprite pointers* are used. There are eight of these, and each can be POKEd with a value from Ø to 255. This value is multiplied by 64 to identify the sprite location. (This is the reason why the memory location has to be a multiple of 64.)

The maximum value—255—gives, therefore, a ceiling value of 255*64, which is 16K, covering the whole of one block of memory which the video chip can access. (There are four such blocks, or 16K *banks*—only one can be accessed at any one time.)

The pointers are a powerful tool which permit you to switch to and fro from any sprite definition which may be in memory. A whole sequence of sprites can be called in this way. Each one will replace its predecessor, so permitting imaginative animation effects.

In fact, this is a much more efficient way of making use of sprites: switching the pointer rather than the sprites themselves leaves the sprites available for other uses.

TROUBLE SHOOTER

TAPE SAVES

One of the unpublicised problems when working with sprites is attempting to SAVE programs on tape. This occurs only if you've made the quite forgivable error of actually RUNning, during the course of its development, a program containing a sprite. Quite normal, you might think, but a source of the mystifying SAVE errors rather too easily blamed on the C2N cassette unit!

The *only* fail-safe solution to the problem is to precede your normal SAVE command with the following POKE:

```
POKE 53269,Ø:SAVE "PROGNAME"
```

This disables the sprite display and eliminates the source of interference which affects the SAVE routine on tape. As an alternative, add the POKE to the end of the program itself.

The same POKE should be used to precede a LOAD instruction when a new program is LOADed into the Commodore following a sprite display. This is not necessary if you effect a system *coldstart*—usually done simply by switching the machine off and then on!

Disk users will be happy to know that sprite-related SAVE and LOAD problems do not extend to these units, and normal procedures may be used.

The sprite pointers are always located at the unused end eight bytes of the 1K screen display memory, which is normally from 2Ø4Ø to 2Ø47. You can see an example of this in Line 3Ø of the flying bird program, where 2Ø4Ø is used.

It is possible to relocate the screen memory and in so doing move the sprite pointers too. Fresh values for the pointers have to be POKEd into the new sprite pointer locations.

WHERE THE ACTION IS

Before you can control the sprite, you need to come to grips with the workings of the 6566 VIC-II chip used by the Commodore 64. In particular, you need to know how to access the control commands.

This isn't as fearsome as it sounds and is covered in greater detail in a later article. For the moment, take a glance at the following table, which lists the 47 memory addresses from 53,248 to 53,294 that are used in sprite programming. The commonly used V + value shows the relationship of all related addresses, based on the starting address where V = 53248.

At first sight the table seems rather daunting but it is nevertheless an essential reference point if you're going to get the most from your sprite programming. The V + value notation is rather easier to remember than the specific addresses, and is more efficient on memory usage—this would be especially beneficial in long programs. Additionally, their use considerably simplifies identification of specific sprite instructions—a useful aid when program changes and debugging are necessary.

You can see that each of the normal eight sprites is numbered, and the value of the number has important consequences.

You can also see from the table that there are locations for controlling everything from sprite positioning to collision detection and colour.

The actual use of V + value notation is described in much greater detail in the next part of this article which uncovers the inner workings of sprite control. But if you're curious, have a look at the use of V + values in the bird program.

There in Line 25 is V + 21, the enabling command which turns on the sprite. And in Line 7Ø the V values set up the X and Y positions of the sprite on the screen. Further examples which you can analyze by referring to the table appear in Line 24ØØ and others in the Space Station program.

VIC-II chip memory locations

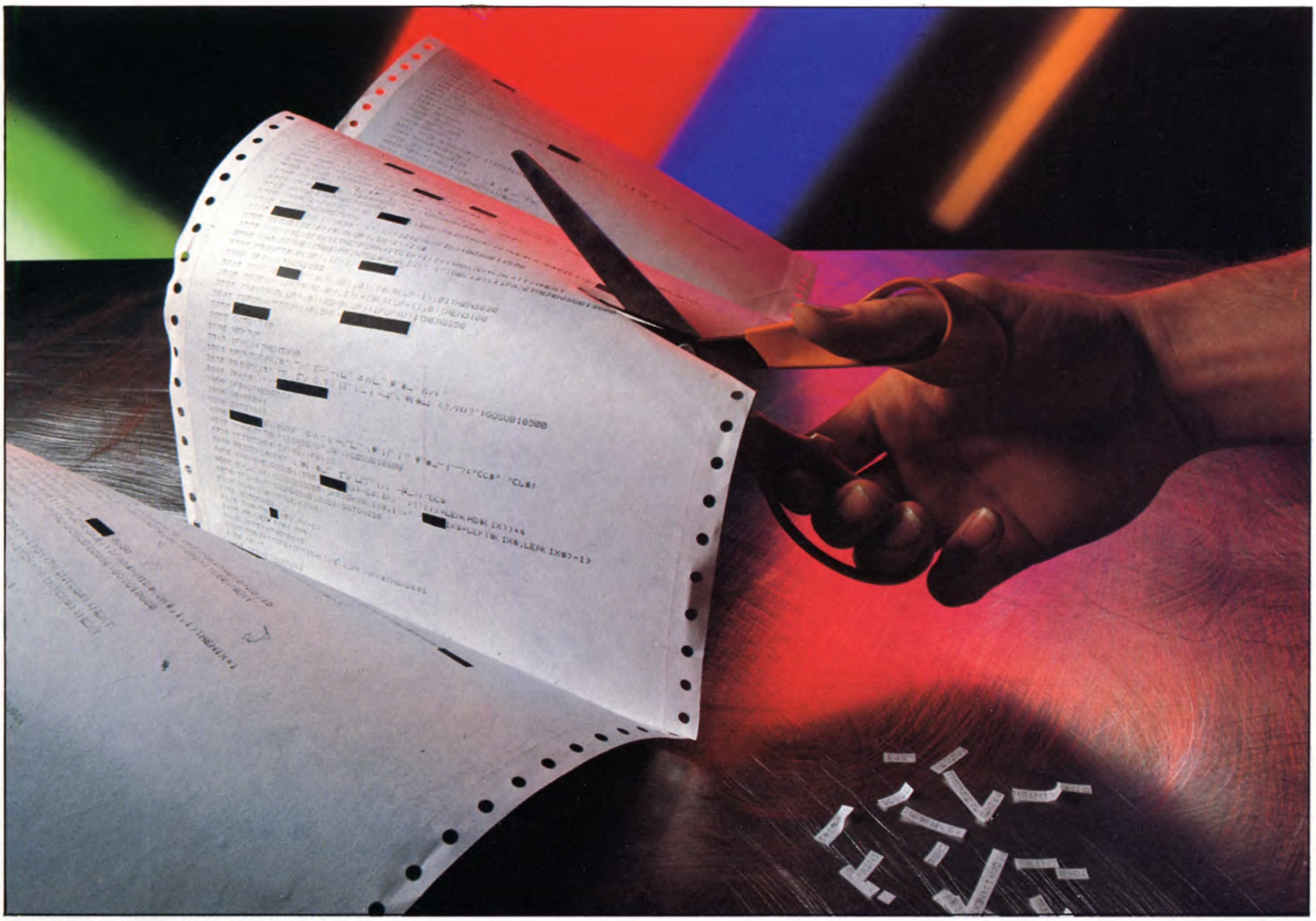
This is a handy reference table of the memory locations you will need to access to control the shape, size, colour and position of your sprites.

The V + value numbers are the most useful and the easiest to remember.

Decimal	V + value	Description
53248	V	Sprite-Ø X position
53249	V+1	Sprite-Ø Y position
5325Ø	V+2	Sprite-1 X position
53251	V+3	Sprite-1 Y position
53252	V+4	Sprite-2 X position
53253	V+5	Sprite-2 Y position
53254	V+6	Sprite-3 X position
53255	V+7	Sprite-3 Y position
53256	V+8	Sprite-4 X position
53257	V+9	Sprite-4 Y position
53258	V+1Ø	Sprite-5 X position
53259	V+11	Sprite-5 Y position
5326Ø	V+12	Sprite-6 X position
53261	V+13	Sprite-6 Y position
53262	V+14	Sprite-7 X position
53263	V+15	Sprite-7 Y position
53264	V+16	MSB of X coordinate
53265	V+17	VIC control register
53266	V+18	Raster register
53267		(light pen)
53268		(light pen)
53269	V+21	Sprite display enable
5327Ø	V+22	VIC control register
53271	V+23	Sprite Ø-7 Y-expand
53272	V+24	VIC memory control
53273	V+25	Interrupt register
53274	V+26	Interrupt enable
53275	V+27	Background priority
53276	V+28	Select multicolour
53277	V+29	Sprite Ø-7 X-expand
53278	V+3Ø	Sprite collision
53279	V+31	Background collision
5328Ø	V+32	Screen border colour
53281	V+33	Background colour Ø
53282	V+34	Background colour 1
53283	V+35	Background colour 2
53284	V+36	Background colour 3
53285	V+37	Sprite multicolour 1
53286	V+38	Sprite multicolour 2
53287	V+39	Sprite-Ø colour
53288	V+4Ø	Sprite-1 colour
53289	V+41	Sprite-2 colour
5329Ø	V+42	Sprite-3 colour
53291	V+43	Sprite-4 colour
53292	V+44	Sprite-5 colour
53293	V+45	Sprite-6 colour
53294	V+46	Sprite-7 colour

GET YOUR PROGRAMS IN SHAPE

- WHAT IS STRUCTURED PROGRAMMING?
- USING FLOW CHARTS
- HOW TO USE STRUCTURES IN BASIC PROGRAMS



Good design makes your programs easier to understand and easier to work on. It can also make the difference between them RUNning efficiently or failing dismally

When you decide to write your first program for a computer there is usually an uncontrollable urge to sit down at the keyboard and start typing in some part of the program immediately. Maybe the first few lines you write work. You feel pleased with yourself and add a few more. You get these to work as well and so you go on adding lines here and there, testing as you go until the program has expanded to perhaps a few hundred lines.

Then, suddenly, disaster! You add a few lines of code and it does not work any more. You cannot see any possible reason why it has stopped working. You change this line, that line, but still no joy.

However, do not give up. There are ways to stop this happening and all that it takes is a little organization. If you follow a few simple rules when you write your programs then you should have no trouble at all.

STRUCTURED PROGRAMMING

Standard BASIC was designed to be easy to use but it has very few *structures* compared to other languages and that makes it more difficult to write structured programs. Structures are the building blocks that you use to construct the main shape of your program. In standard BASIC they are IF . . . THEN, FOR . . . NEXT, GOTO and GOSUB, and in BBC BASIC there are three extra ones—REPEAT . . . UNTIL, PROCedures and functions. You've seen how most of these are used on their own but the idea now is to put them together in an orderly and readable way.

It is very easy and quick to write a few lines of program that work first time (almost) and are relatively simple for someone else to understand. In fact it's not important to take special care structuring the program if it is short. The problem comes when you want to write a large program that does something useful. In this case, if you start by writing a program fragment and then add more and more of the program to it you are likely to get hopelessly lost—unless you have the memory of an elephant, that is.

What you need to do is sit down and systematically design the program first. Most programs start with a vague idea of something you wish the computer to do—and the more complex this is, the vaguer your ideas are likely to be. The basic concept may be for a game or a word processing package or anything of a similar type which is so large that you cannot hold all the ideas about it in your head at once.

Start by finding somewhere well away from the computer—preferably in another room to avoid temptation—and write down what you want to do in a very general way. This is the beginnings of what is known as a *design specification* or *spec* for short. For example you might write:

Index system
The program should allow records in memory to be created, updated, deleted, sorted and listed. The records are to be accessed using a keyword. The entire set of records may be SAVEd into and LOADed from a file.

Next you must break down this very general description into a few logical steps or *modules*. The operations involved in each module will probably still be very complicated so they must be broken down into smaller sections until you think that each of the lowest level of modules is short enough to handle. **Fig. 1** shows how this might be done for the Index System.

Each of the smallest sections should be no more than a page in length—about 6Ø lines—but half this length is even better. Each of these low level modules should be very easy to understand. Eventually each module will end up as a subroutine in your program.

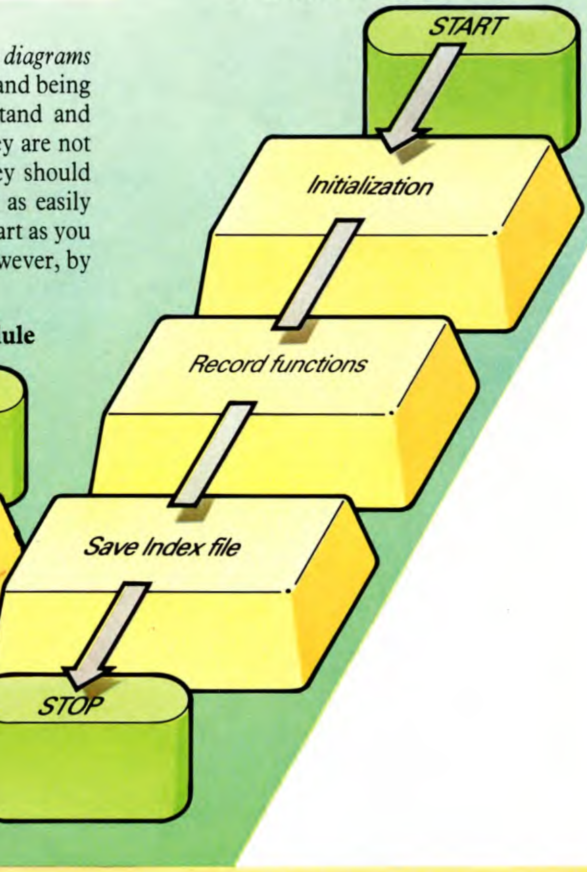
The process of breaking problems down will become easier with experience. And, as always, the best way to learn how to do it is to try it. This method of breaking up the problem is known as top-down design. You started at the top (the general description of the program) and worked down to the bottom (the lowest level of modules). However, you still have not decided on the order of the modules—that is, the order in which they are executed by the program. So this comes next.

FLOW DIAGRAMS

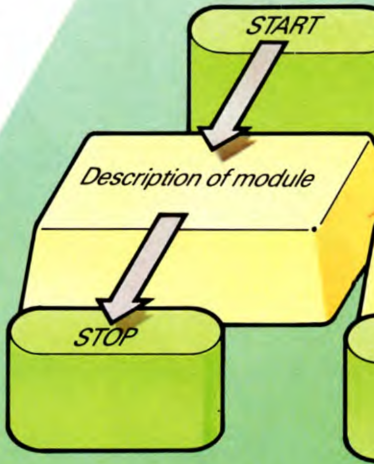
One way to specify this is to use *flow diagrams* or *flow charts*. These are easy to use and being diagrammatical are easy to understand and follow at a glance. Like BASIC, they are not in themselves structured and so they should be used with caution. You can just as easily have a tangled and confused flow chart as you can have a jumbled up program. However, by following a simple set of rules they may be used to sort out a program in a straightforward and structured way.

Fig. 2 shows how a single module can be specified. The program follows along the lines in the direction of the arrows and the boxes describe what happens at each stage. To execute a series of modules in sequence you just add on more boxes in the right order between start and stop (see **fig. 3**).

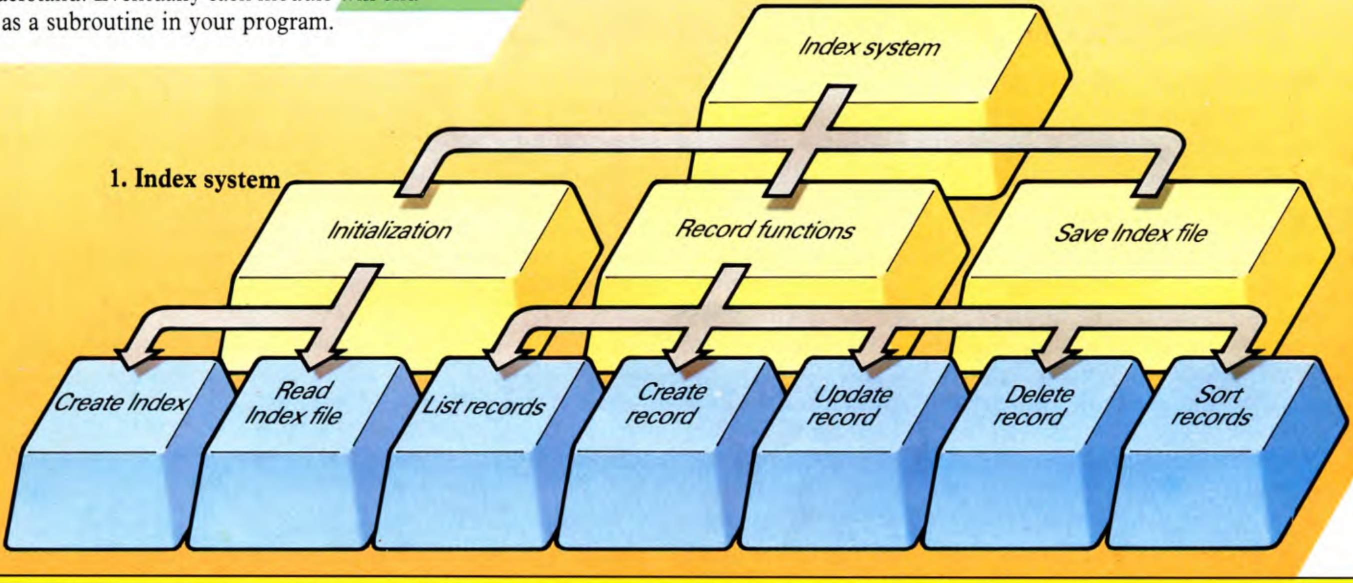
3. Combining several modules



2. A single module



1. Index system



A simple flow diagram like this is fine for a program in which no decisions are made. However, the power of the computer depends heavily on its ability to make choices within a program. This is the familiar IF ... THEN statement and it is about the most often used structure in BASIC.

Let's have a look now at all the different structures and how you can represent them with flow diagrams.

IF ... THEN ... ELSE

Although there are differences between the way different computers use the command, IF ... THEN ... ELSE is the basis of all decisions that the computer has to make. Its flow diagram is shown in **fig. 4** and in BASIC it is written as:

```
1ØØ IF condition THEN statement1 ELSE
   statement2
```

It means, if the condition is true then statement 1 is executed, otherwise statement 2 is executed.

If you look at the flow diagram you'll notice that there is only one entry point and one exit. This greatly aids testing and debugging since you know exactly where this section of the program must start and finish. In fact this is a very important rule of structured programming: for any section of code there should be only one entry and one exit.

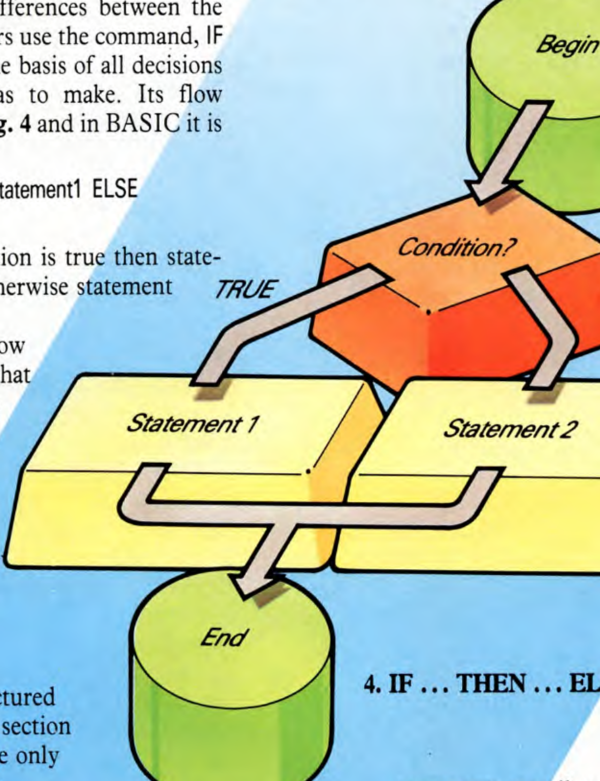
Most versions of BASIC do not have the ELSE part—it isn't available on the Spectrum, ZX81, Commodore 64 or Vic. But that doesn't matter too much as it can be simulated using GOTO:

```
1ØØ ...
11Ø IF condition THEN GOTO 14Ø
12Ø statement2 :REM this is the ELSE part
13Ø GOTO 15Ø
14Ø statement1 :REM this is the THEN part
15Ø ...
```

Of course the line numbers need not be as shown. Additionally, more than one statement may be included in the THEN and ELSE parts. For example, here's a section of program to sort two numbers into the correct order. It is the basis of an alphabetical sort routine which will be given in Part 2 of this article. The ELSE part in this case has four statements:

```
1ØØ IF first< =second THEN GOTO 16Ø
11Ø LET temporary=first
12Ø LET first=second
13Ø LET second=temporary
14Ø LET order$="wrong"
15Ø GOTO 17Ø
16Ø LET order$="right"
17Ø ...
```

This example can also be written out using ELSE but then it must all be written on one line. Multiple statements are allowed as long as they are separated by colons:



4. IF ... THEN ... ELSE

```
1ØØ IF first>second THEN temporary=
   first: first=second: second=temporary:
   order$="wrong" ELSE order$="right"
```

As you can see, it's not easy to read or understand programs written using long statements like this. So they should be avoided if at all possible.

Finally, in many cases you may not need the ELSE part at all. The flow diagram looks like **fig. 5** and it is written out as:

```
1ØØ IF condition THEN statement
```

which, of course, is just the simple IF ... THEN statement.

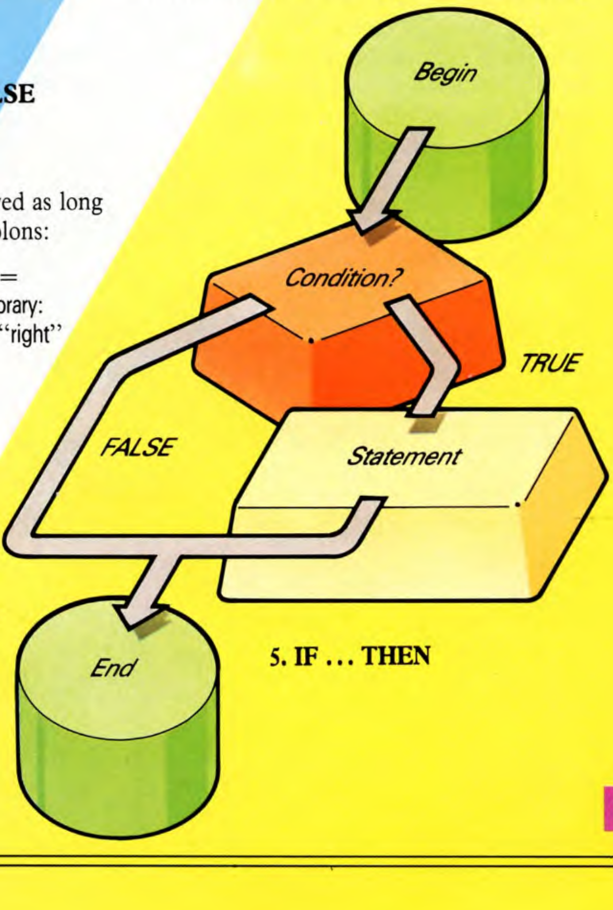
NESTED STRUCTURES

The IF ... THEN ... ELSE lines may be nested. That is, one or both of the statements between which the IF ... THEN line chooses may themselves be an IF ... THEN line. For example, this section of program keeps tally of how many games two players have won, and PRINTs out the results after each game:

```
1ØØ IF T1< >T2 THEN GOTO 13Ø
11Ø PRINT "It's a draw!"
12Ø GOTO 19Ø
13Ø IF T1<T2 THEN GOTO 17Ø
14Ø PRINT "The first player wins"
15Ø LET P1=P1+1
16Ø GOTO 19Ø
17Ø PRINT "The second player wins"
18Ø LET P2=P2+1
19Ø ...
```

All structures can be nested in any combination and, in theory, to any depth. However, the more nesting you have, the less readable the program becomes and a sensible limit is a depth of three or possibly four structures. If you find yourself needing more then you should split the program into smaller modules or subroutines.

Have a look at the last program again. It is actually rather difficult to follow what is going on even though it is perfectly well structured. A useful way to make nested



5. IF ... THEN

statements more readable is to indent the program lines. This is only possible on the Acorn and Spectrum computers but on these you can rewrite the last program as follows:

```
100 IF T1< >T2 THEN GOTO 130
110  PRINT "It's a draw!"
120 GOTO 190
130  IF T1<T2 THEN GOTO 170
140   PRINT "The first player wins"
150   LET P1=P1+1
160  GOTO 190
170   PRINT "The second player wins"
180   LET P2=P2+1
190 ...
```

On the Acorn computers you can automatically indent all the structured statements by typing LISTO 7 followed by LIST.

Other ways of making the structure of your program clearer are by inserting blank lines around different sections and by using REM statements. Blank lines can be entered on the Spectrum and Acorn computers by typing the line number followed by a space and then [ENTER] or [RETURN]. You can get a similar effect to this when you are writing programs on the Dragon, Tandy and Commodore by typing a colon instead of the space.

WHILE ... DO

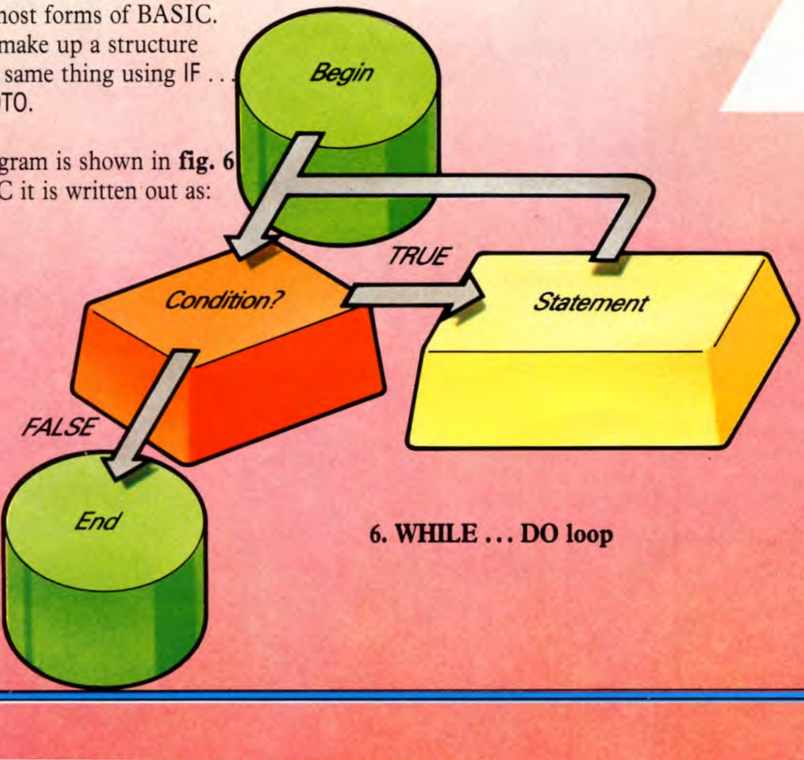
The other essential structure is the WHILE ... DO statement. This allows looping in a program and is one of the most useful ways of creating a loop. It makes the computer DO something over and over again WHILE a certain condition is true. Don't worry that your computer doesn't have the words WHILE and DO. They are not available in most forms of BASIC. But you can make up a structure that does the same thing using IF ... THEN and GOTO.

The flow diagram is shown in **fig. 6** and in BASIC it is written out as:

```
100 ...
110 IF NOT(condition) THEN GOTO 140
120 statement
130 GOTO 110
140 ...
```

Notice that Line 110 reads IF NOT(condition) ... That means it is checking for when the condition is NOT true—the opposite to normal. But there is no problem. If the condition is A=B then NOT(A=B) is A< >B—similarly NOT(A<B) is A>=B and so on. You can actually write NOT(A=B) if you like and the computer will understand what you mean.

Here is an example of a short program using a WHILE loop that you can use to time an egg:



6. WHILE ... DO loop

```
5 CLS
10 PRINT AT 3,11;"EGG TIMER"
20 INPUT "How many minutes do you
   want?",t
30 PRINT AT 7,5;"Press any key to start"
40 PAUSE 0
50 CLS
60 PRINT FLASH 1;AT 10,9;
   "□T I M I N G□"
70 POKE 23672,0: POKE 23673,0
80 LET time=PEEK 23672+256*PEEK
   23673: IF time>t*50*60 THEN
   GOTO 110
90 PRINT AT 14,10;INT (time/50); "□seconds"
100 GOTO 80
105 REM end of WHILE loop
110 PRINT FLASH 1;AT 14,10;"□IT'S
   DONE□"
120 BEEP .5,20
```

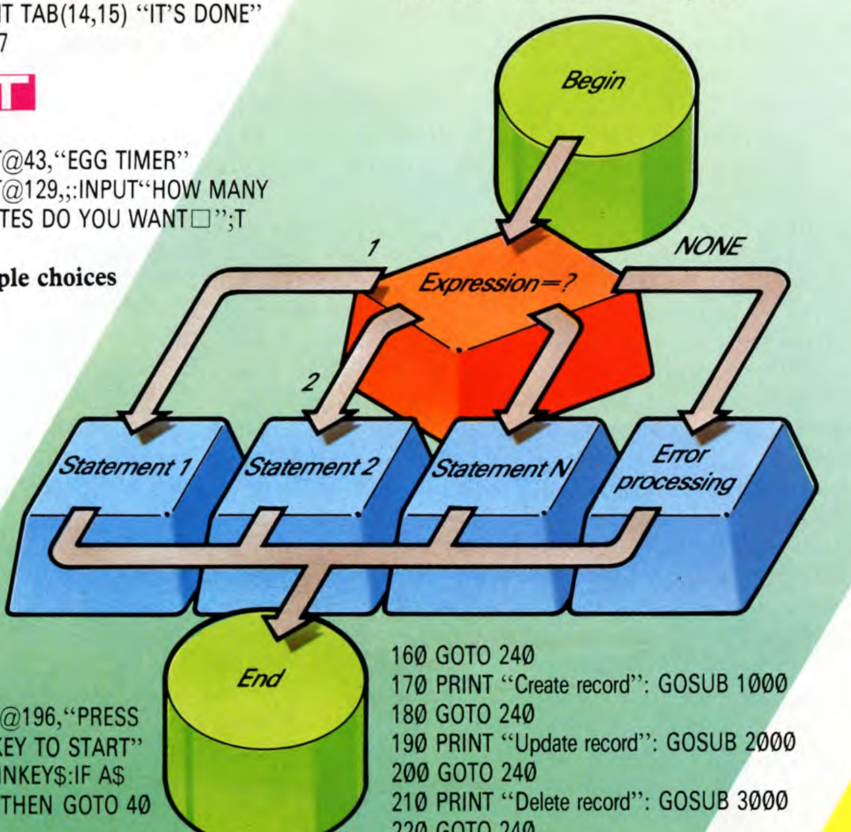
```
5 PRINT "♥■":POKE 53281,4
10 PRINT TAB(16)"▨EGG TIMER"
20 INPUT "▨▨▨□□□□□
   HOW MANY MINUTES DO YOU WANT";T
30 PRINT TAB(8)"▨▨▨PRESS SPACE
   BAR TO START"
40 GET A$:IF A$="" THEN GOTO 40
50 PRINT "♥"
60 PRINT TAB(15)"T I M I N G"
70 TI$="000000"
75 REM START OF WHILE LOOP
80 IF VAL(TI$)=>T*100 THEN GOTO 110
90 PRINT "▤▨▨▨▨▨"TAB(15);
   RIGHT$(TI$,2)"□SECONDS"
100 GOTO 80
105 REM END OF WHILE LOOP
110 PRINT TAB(16)"▨▨▨IT'S DONE"
120 POKE 54296,15:POKE 54278,128:
   POKE 54276,17:POKE 54273,50
130 FOR D=1 TO 200: NEXT:POKE
   54276,0:POKE 54278,0
```

```
5 PRINT "♥■":POKE 36879,29
10 PRINT TAB(7)"▨EGG TIMER"
20 PRINT "▨▨▨HOW MANY MINUTES
   DO":INPUT"□YOU WANT";T
30 PRINT "▨▨▨ HIT SPACE BAR TO
   START"
40 GET A$:IF A$="" THEN 40
50 PRINT "♥"
60 PRINT TAB(8)"TIMING"
70 TI$="000000"
75 REM START OF WHILE LOOP
80 IF VAL(TI$)=>T*100
   THEN 110
90 PRINT "▤▨▨▨▨▨"TAB(6);
   RIGHT$(TI$,2)"□SECONDS"
100 GOTO 80
105 REM END OF WHILE LOOP
110 PRINT TAB(6)"▨▨▨
   IT'S DONE"
120 POKE36878,15:POKE36876,200
130 FOR D=1 TO 200:NEXT:POKE
   36876,0
```

```
5 CLS
10 PRINT TAB(15,2) "EGG TIMER"
20 INPUT TAB(5,4) "HOW MANY MINUTES
   DO YOU WANT□",T
30 PRINT TAB(7,6) "PRESS SPACE BAR
   TO START"
40 K$=GET$
50 CLS:VDU23;8202;0;0;0;
60 PRINT TAB(15,10) "T I M I N G"
```

```
70 TIME=Ø
75 REM start of WHILE loop
80 IF TIME>T*6Ø*1ØØ THEN
   GOTO 11Ø
9Ø PRINT TAB(13,12);INT(TIME/1ØØ);
   "□seconds"
1ØØ GOTO 8Ø
1Ø5 REM end of WHILE loop
11Ø PRINT TAB(14,15) "IT'S DONE"
12Ø VDU7
```

```
5 CLS
1Ø PRINT@43,"EGG TIMER"
2Ø PRINT@129,;:INPUT"HOW MANY
   MINUTES DO YOU WANT□";T
```



7. Multiple choices

```
3Ø PRINT@196,"PRESS
   ANY KEY TO START"
4Ø A$=INKEY$:IF A$
   ="" THEN GOTO 4Ø
5Ø CLS
6Ø PRINT@235,"T I M I N G"
7Ø TIMER=Ø
75 REM START OF WHILE LOOP
8Ø IF TIMER>T*3ØØØ THEN GOTO 11Ø
9Ø PRINT@298,INT(TIMER/5Ø);"SECONDS"
1ØØ GOTO 8Ø
1Ø5 REM END OF WHILE LOOP
11Ø PRINT@364,"IT'S DONE"
12Ø SOUND 18Ø,3
```

MULTIPLE CHOICES

The IF . . . THEN and WHILE . . . DO structures are usually sufficient for most programs. However, there are a few extra structures to make programming even easier. For example, there are often more than two courses of action possible at a particular point in a program. This could be catered for using nested IF . . . THENs but it is more convenient to use what is known as a CASE structure. This sets up each option in turn and directs the computer to a series of possible courses.

The flow diagram for a CASE structure is shown in **fig. 7** and in BASIC a typical program would look like this:

```
1ØØ REM CASE of command
11Ø IF C$="C" THEN GOTO 17Ø
12Ø IF C$="U" THEN GOTO 19Ø
13Ø IF C$="D" THEN GOTO 21Ø
14Ø IF C$="L" THEN GOTO 23Ø
15Ø PRINT "Command not recognised"
16Ø GOTO 24Ø
17Ø PRINT "Create record": GOSUB 1ØØØ
18Ø GOTO 24Ø
19Ø PRINT "Update record": GOSUB 2ØØØ
2ØØ GOTO 24Ø
21Ø PRINT "Delete record": GOSUB 3ØØØ
22Ø GOTO 24Ø
23Ø PRINT "List records": GOSUB 4ØØØ
24Ø REM end of CASE
```

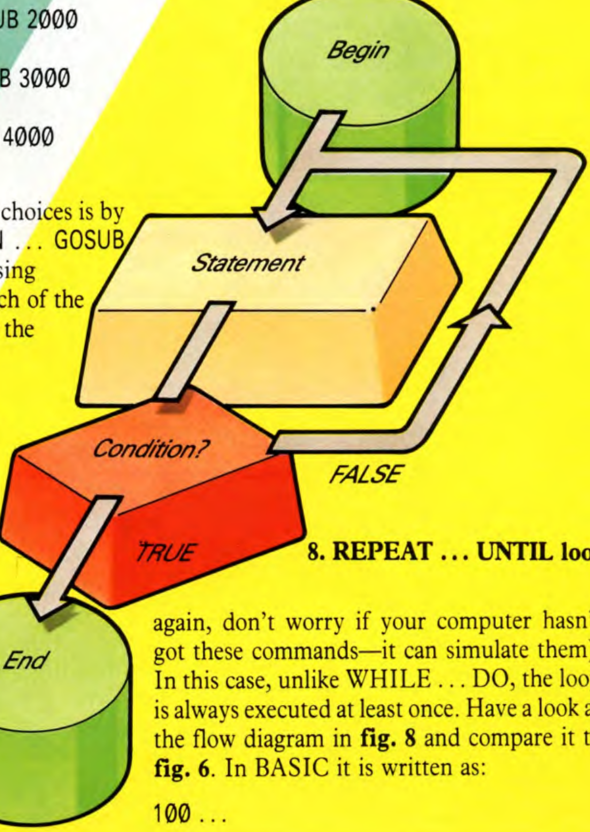
Another way of making multiple choices is by using the ON . . . GOTO and ON . . . GOSUB statements. However, when using ON . . . GOTO, make sure that each of the options eventually directs you to the end of the section, as shown in the next example where a GOTO 121Ø is needed after each option to direct the program to the end of the routine.

```
1ØØØ REM POLYGON
   SUBROUTINE
1Ø1Ø INPUT "HOW MANY
   SIDES DO YOU WANT";N
1Ø2Ø ON N-2 GOTO
   1Ø6Ø,11ØØ,114Ø,
   116Ø,118Ø,12ØØ
1Ø3Ø PRINT "I DON'T
   KNOW THE
   NAME OF A"
1Ø4Ø PRINT "POLYGON
   WITH□";N;
   "□SIDES."
1Ø5Ø GOTO 121Ø
1Ø6Ø PRINT "THAT'S A TRIANGLE."
1Ø7Ø PRINT "A TRIANGLE WITH EQUAL
   SIDES IS"
1Ø8Ø PRINT "CALLED AN EQUILATERAL
   TRIANGLE."
1Ø9Ø GOTO 121Ø
11ØØ PRINT "THAT'S A QUADRILATERAL."
111Ø PRINT "A QUADRILATERAL WITH
   EQUAL SIDES"
112Ø PRINT "AND ANGLES IS CALLED
   A SQUARE."
113Ø GOTO 121Ø
114Ø PRINT "THAT'S A PENTAGON."
115Ø GOTO 121Ø
116Ø PRINT "THAT'S A HEXAGON."
117Ø GOTO 121Ø
118Ø PRINT "THAT'S A HEPTAGON."
119Ø GOTO 121Ø
12ØØ PRINT "THAT'S AN OCTAGON."
121Ø PRINT
122Ø RETURN
```

This is just a subroutine so you can't RUN it yet. The program to call the routine is given in the next section.

REPEAT . . . UNTIL

The REPEAT . . . UNTIL statement is another useful way of making a program loop (once again, don't worry if your computer hasn't got these commands—it can simulate them). In this case, unlike WHILE . . . DO, the loop is always executed at least once. Have a look at the flow diagram in **fig. 8** and compare it to **fig. 6**. In BASIC it is written as:



8. REPEAT . . . UNTIL loop

```
1ØØ . . .
```

```
11Ø statement
12Ø IF NOT(condition) THEN GOTO 11Ø
13Ø . . .
```

Using the subroutine from the last example you can write a program using a REPEAT loop as follows:

```
1Ø PRINT "I WILL TELL YOU THE NAMES"
2Ø PRINT "OF SOME POLYGONS."
3Ø REM START OF LOOP
4Ø GOSUB 1ØØØ
5Ø INPUT "DO YOU WANT ANOTHER
   NAME□";A$
6Ø IF LEFT$(A$,1) = "Y" THEN GOTO 3Ø
7Ø PRINT "GOODBYE!":END
```

Line 1ØØØ is the polygon subroutine given in the last section.

Note that on some BBC computers you may need to change the semicolon to a comma in Line 5Ø of the program and Line 1Ø1Ø of the subroutine.

BBC BASIC actually has a REPEAT ... UNTIL statement. What is more, the UNTIL part of the statement need not be on the same line as the REPEAT part, unlike the case of the IF ... THEN ... ELSE statement. So the last example can be written as:

```
1Ø PRINT "I will tell you the names"
2Ø PRINT "of some polygons."
3Ø REPEAT
4Ø GOSUB 1ØØØ
5Ø INPUT "Do you want another name□",
   A$
6Ø UNTIL LEFT$(A$,1) < > "Y"
7Ø PRINT "Goodbye!"
```

This is a lot easier to follow than the last version.

FOR ... NEXT LOOPS

You may not have realized it, but the familiar FOR ... NEXT loop is just a special case of the WHILE ... DO loop. It should be used when the number of times round the loop is known beforehand as this has to be specified at the start. The variable that keeps count of the number of times round the loop is known as the *control variable.*

A flow chart for a FOR ... NEXT loop looks something like **fig. 9.** Compare it to the WHILE loop in **fig. 6** and you'll see it has the same overall structure. In BASIC it is written:

```
1ØØ FOR i = min TO max STEP val
11Ø statement
12Ø NEXT i
```

It is bad programming practice to jump out of a FOR loop using a GOTO statement since there is no way that the BASIC interpreter can know you have done this. You could quite legally jump back into the loop later on in the program, but understanding such a program would be hard. So don't do it!

PUTTING IT ALL TOGETHER

These structures are all you need, whatever type of program you are writing. But there is still a lot more to do before you can complete the program. You have to specify the variables you are going to use and make sure that the variables in the different modules don't clash with each other. Then you have to sort out what inputs and outputs are needed by the program.

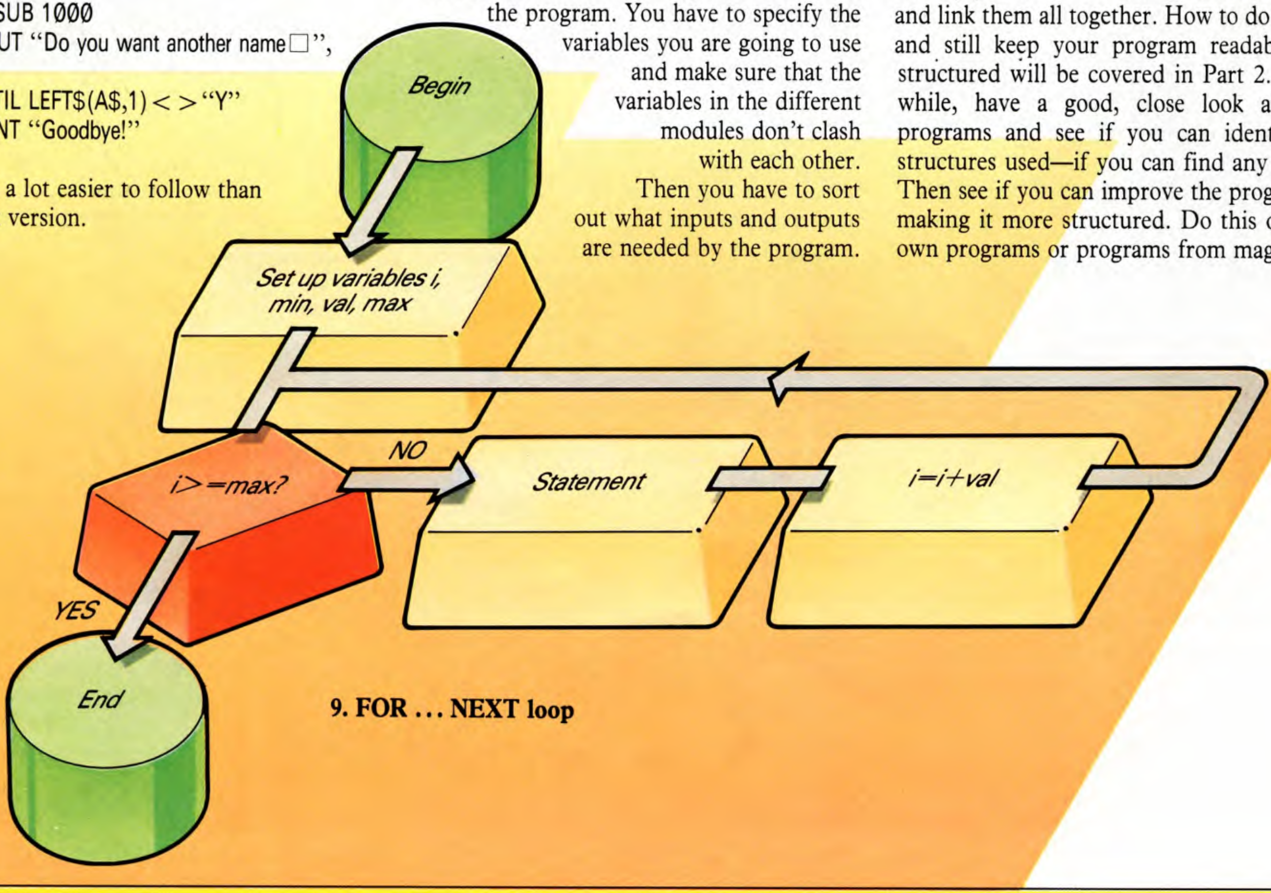
Q+A

Does the shape of the boxes used in flowcharts have any special significance?

Yes, flowcharts are drawn using standard symbols. There are five main shapes:

1. Rounded oblongs are called *terminal* boxes and they show where a program begins and ends.
2. Circles are *connecting* symbols used at the beginning and end of a module.
3. Rectangles are *instruction* boxes containing program statements.
4. Diamond shapes are *decision* boxes. There are always at least two routes out of these boxes depending on a decision made inside the box.
5. Finally, parallelograms (not shown in this article) are *input/output* boxes indicating any information input to the program, and any output to the screen or a printer.

Finally you have to test each of the modules and link them all together. How to do all this and still keep your program readable and structured will be covered in Part 2. Meanwhile, have a good, close look at some programs and see if you can identify the structures used—if you can find any that is! Then see if you can improve the program by making it more structured. Do this on your own programs or programs from magazines.



9. FOR ... NEXT loop

GETTING DOWN BELOW ZERO

- WHEN YOU NEED NEGATIVE NUMBERS
- NEGATIVE NUMBER CONVERSION PROGRAM
- THE SIGN CONVENTION

Binary and hex are easy to understand, and they translate directly onto the computer. But both have problems if you need to represent negative numbers.

In games programming you will find that you need to use negative numbers, for example, to move about the screen in certain directions. These have to be encoded in eight-bit bytes, since the computer has no other means to store anything in its memory. And this raises a problem: as you have seen, a byte can represent any number between Ø and 255, or ØØØØØØØØ and 11111111. But this uses up all the binary possibilities in eight bits—there is no room for a minus or plus sign and no way in which it could be represented.

In ordinary arithmetic, if you subtract 1 from Ø you get −1. So try that calculation in eight-bit binary:

```
ØØØØØØØØ
      −1
--------
11111111
```

This does leave you trying to 'borrow' one from the next place to the left—but when the binary number is limited to eight bits there is nowhere left to borrow one from.

Similarly, if you subtract another 1 (to give −2 in ordinary arithmetic) you get 1111111Ø. But if 11111111 in binary is 255 in decimal, then 1111111Ø must be 254!

It isn't just in binary that you can get oddities like this. Imagine, for example, what would happen in decimal if you had no more than three places, or 'columns', in which to put your numbers. Look what happens when

you try to add 999 to 1ØØ when these limitations are imposed:

```
  1ØØ
+ 999
(1)Ø99
```

The 1 in the 'thousands' place has had to go in brackets. In our three-place arithmetic, there is just no room for it. So the result of this addition, in the three-place system, is 99—which is exactly what you get if you subtract 1 from 1ØØ!

Similarly, in a three-place system adding 998 to 1ØØ will act like subtracting 2. And subtracting 998 from 1ØØ will act like adding 2.

To sum up: the same row of numbers in an eight-bit byte can represent either a negative or a positive number—on the face of it, highly confusing and hard to work with.

What can you do about it? Well, for most home computer applications there is no need to worry. Memory addresses and operation codes, which are both rendered in binary, can be considered positive always. The only times you will come across negative numbers are in DATA or jumps, which are the machine code equivalent of GOTO statements.

FLIPPING THE BITS

In binary the process used to get from a positive number to its negative value is known as *2's complement*. It has no really solid theoretical basis—or at least none that is easy to understand—but it works.

To get from one binary number to its negative value, you flip the bits and add 1. 'Flip the bits' means change the bits that are 1 to Ø and those that are Ø to 1.

The following program shows you how this works. Note that when binary is limited to eight digits its hex equivalent exactly fills two digits. This means that the hex equivalent of the 2's complement also acts as the negative.

```
10 PRINT AT Ø,7;"NEGATIVE NUMBERS"
2Ø PRINT AT 2,1;"DEC";TAB 14;
   "BIN";TAB 28;"HEX"
3Ø LET A$="□□□□□□□□
   □□□□□□□□□□□"
4Ø FOR N=7 TO 13
5Ø PRINT AT N,6; INVERSE 1;A$
6Ø NEXT N
7Ø PRINT AT 8,8;"FLIP BITS";
   BRIGHT 1;AT 12,21;"+1"
8Ø PRINT AT 1Ø,3;"+";AT 1Ø,3Ø;"+"
9Ø PRINT AT 15,8;"2'S COMPLEMENT"
95 LET C=Ø
1ØØ LET DD=-C: DIM A(8)
11Ø PRINT AT 4,Ø;"□□□□";AT
   4,4-LEN STR$ C;C
115 POKE 236Ø8,C: LET E=PEEK 236Ø8:
   LET Z=E: GOSUB 3ØØ: PRINT AT
   4,29;A$
12Ø PRINT AT 17,Ø;"□□□□";AT
   17,4-LEN STR$ DD;DD
13Ø LET D=128: LET CC=E
14Ø FOR N=1 TO 8: LET A(N)=Ø
15Ø IF CC-D>=Ø THEN LET A(N)=1: LET
   CC=CC-D
16Ø PRINT BRIGHT 1;AT 4,6+2*N;A(N);
   AT 1Ø,6+2*N;1-A(N)
17Ø LET D=D/2: NEXT N
18Ø POKE 236Ø8,DD: LET DD=PEEK
   236Ø8: LET D=128
185 LET Z=DD: GOSUB 3ØØ: PRINT
   AT 17,29;A$
19Ø FOR N=1 TO 8: LET B=Ø: IF
   DD-D>=Ø THEN LET B=1: LET
   DD=DD-D
2ØØ PRINT BRIGHT 1;AT 17,6+2*N;B:
   LET D=D/2: NEXT N
21Ø PRINT AT 18,Ø;"----□□
   -------------
   ------□□----"
22Ø PRINT AT 19,3;"Ø□□□□
   Ø□Ø□Ø□Ø□Ø□Ø□Ø□Ø
   □□□□□□ØØ"
23Ø IF INKEY$="" THEN GOTO 23Ø
24Ø LET A$=INKEY$: IF A$="□" THEN
   LET C=C+1: IF C=128 THEN LET
   C=-128: BEEP 1,1
25Ø IF A$="B" OR A$="b" THEN LET
   C=C-1: IF C=-129 THEN LET
   C=127: BEEP 1,1
26Ø IF A$<>"□" AND A$<>"B" AND
   A$<>"b" THEN INPUT "?";C
27Ø GOTO 1ØØ
3ØØ LET ZA=INT (Z/16): LET
   ZB=Z-(16*ZA)
31Ø LET ZA=ZA+48: IF ZA>57 THEN
   LET ZA=ZA+7
32Ø LET ZB=ZB+48: IF ZB>57 THEN
   LET ZB=ZB+7
33Ø LET A$=CHR$ ZA: LET A$=A$+CHR$
   ZB: RETURN
```

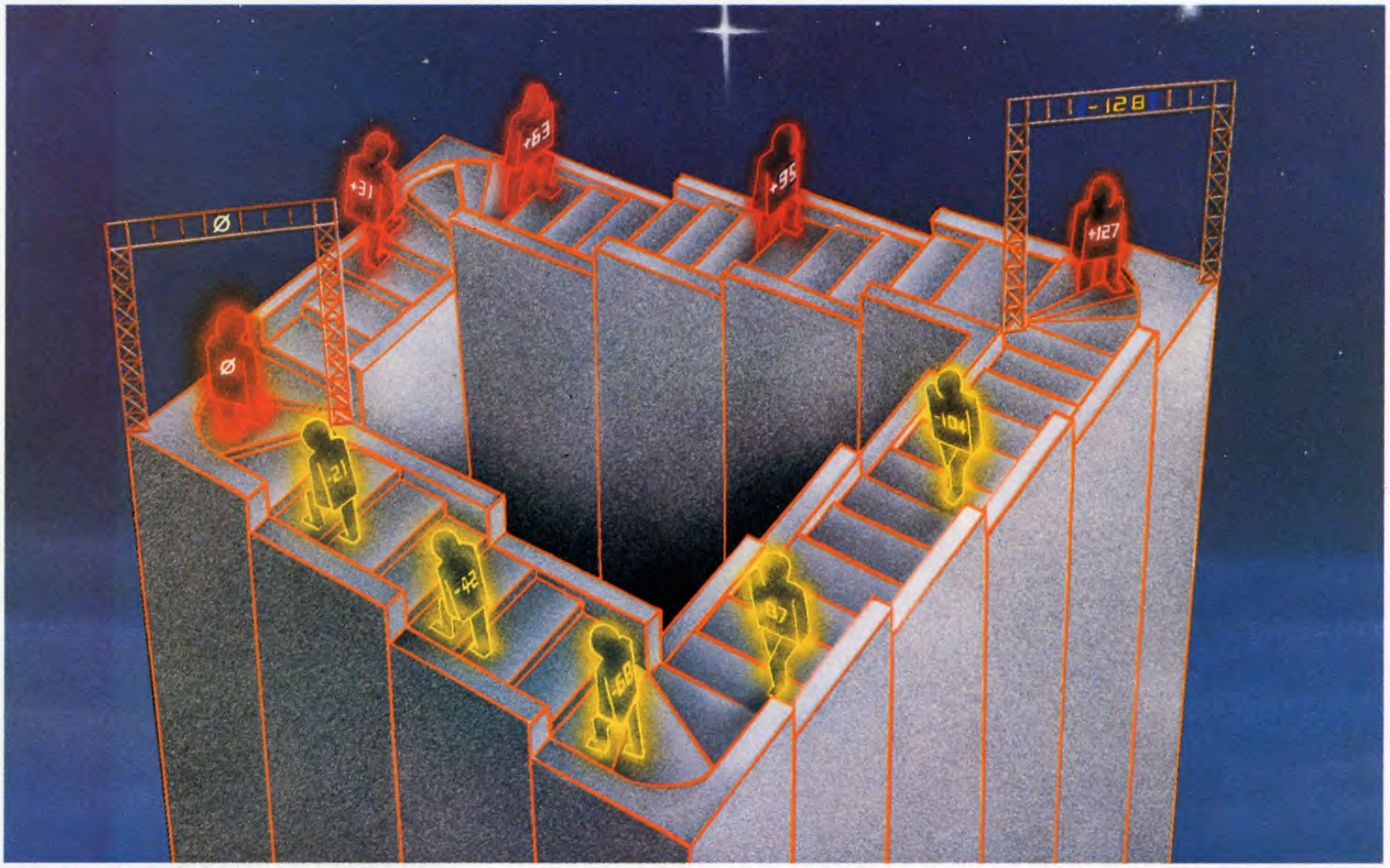
```
1Ø PRINT AT Ø,7;"NEGATIVE NUMBERS"
2Ø PRINT AT 21, 1;"DEC"; TAB 14;
   "BIN";TAB 28;"HEX"
3Ø LET A$="■■■■■■■■■
   ■■■■■■■■■■"
4Ø FOR N=7 TO 13
5Ø PRINT AT N,6; A$
6Ø NEXT N
7Ø PRINT AT 8,8;"FLIP BITS□";AT
   12,21;"+1"
8Ø PRINT AT 1Ø,3;"+"; AT 1Ø,3Ø;"+"
9Ø PRINT AT 15,8;"2S COMPLEMENT"
95 LET C=Ø
1ØØ LET DD=-C
1Ø5 DIM A(8)
11Ø PRINT AT 4,Ø;"□□□□";
   AT 4,4-LEN STR$ C;C
115 POKE 165Ø7, C
116 LET E=PEEK 165Ø7
117 LET Z=E
118 GOSUB 3ØØ
119 PRINT AT 4,29;A$
12Ø PRINT AT 17,Ø;"□□□□";
   AT 17, 4-LEN STR$ DD;DD
13Ø LET D=128
135 LET CC=E
14Ø FOR N=1 TO 8
145 LET A(N)=Ø
15Ø IF CC-D>=Ø THEN LET A(N)=1
155 IF CC-D>=Ø THEN LET CC=
   CC-D
16Ø PRINT AT 4,6+2*N;A(N);AT 1Ø,
   6+2*N; 1-A(N)
17Ø LET D=D/2
175 NEXT N
18Ø POKE 165Ø7,DD
181 LET DD=PEEK 165Ø7
182 LET D=128
185 LET Z=DD
186 GOSUB 3ØØ
187 PRINT AT 17,29;A$
19Ø FOR N=1 TO 8
191 LET B=Ø
```

```
192 IF DD−D>=Ø THEN LET B=1
193 IF DD−D>=Ø THEN LET DD=
    DD−D
200 PRINT AT 17,6+2*N;B
205 LET D=D/2
207 NEXT N
210 PRINT AT 18,Ø;"− − − −
    □□− − − − − − − − − −
    − − − − − − − − −□□
    − − − −"
220 PRINT AT 19,3;"Ø□□□□
    Ø□Ø□Ø□Ø□Ø□Ø□Ø□
    Ø□□□□□□ØØ"
230 IF INKEY$="" THEN GOTO 230
240 LET A$=INKEY$
242 IF A$="F" THEN LET C=C+1
244 IF A$="F" AND C=128 THEN LET
    C=128
250 IF A$="B" THEN LET C=C−1
255 IF A$="B" AND C=−129 THEN LET
    C=127
260 IF A$<>"F" AND A$<>"B" THEN
    INPUT C
270 GOTO 100
300 LET ZA=INT(Z/16)
305 LET ZB=Z−(16*ZA)
310 LET ZA=ZA+28
320 LET ZB=ZB+28
330 LET A$=CHR$ ZA
340 LET A$=A$+CHR$ ZB
350 RETURN
```

```
20 POKE54277,33:POKE54278,255:
   POKE54273+23,15:POKE54276,33:
   POKE54273,Ø
30 FOR Z=1 TO 8:READ A(Z):NEXT Z:
   DATA 128,64,32,16,8,4,2,1
40 Z$="Ø123456789ABCDEF":A=Ø:
   A$="+−":AA=1:POKE65Ø,128:
   V$="▐▐▐▐▐▐▐▐▐"
50 POKE 5328Ø,1:POKE 53281,1:
   PRINT "♡■▄▄▄▄"
60 FOR Z=1 TO 9:PRINT TAB(5)
   "◪□□□□□□□□□□□□□
   □□□□□□□□□□□□□□□
   □□□□□":NEXT Z
70 PRINT "▜▄" TAB(13)"2'S
   COMPLEMENT"
80 PRINT "▀" TAB(12)"NEGATIVE
   NUMBERS"
90 PRINT TAB(12)"− − − − −
   − − − − − − − − − −"
100 PRINT "▩□□DEC□□
    □□□□□□□□□□□□
    BIN□□□□□□□□□□□
    □□□□HEX"
110 PRINT "⌂▄▄▄▄▐▐▐
    ▐▐▐▐▐FLIP BITS":
    PRINT"▄▄▄"TAB(29)"+1"
120 PRINT"○○○○▐▐+"
    TAB(37)"+"
130 PRINT"▄▄▄▄▄▄
    ▄▄▄− − − − − − − − − − −
    − − − − − − − − − − − − −
    − − − − − − − − − − − − −
    − − − − − − − −";
140 PRINT"□□Ø□□□□
    □□Ø□□Ø□□Ø□□Ø□□Ø□
    □Ø□□Ø□□Ø□□□□□□□
    ØØ":GOTO3ØØ
150 GET K$:IF K$=""THEN 15Ø
160 S=Ø:SS=Ø:IF K$<>"□" AND
    K$<>"B" THEN 5ØØ
170 A1=A:IF K$<>"□" THEN 21Ø
180 IF AA=2 THEN A1=A1−1
190 IF AA=1 THEN A1=A1+1
200 GOTO 24Ø
210 IF K$<>"B" THEN 24Ø
220 IF AA=2 THEN A1=A1+1
230 IF AA=1 THEN A1=A1−1
240 IF A1<Ø OR A1>128 THEN
    AA=AA+1
250 IF A1>128 THEN A1=127
260 IF A1<Ø THEN A1=1
270 A=A1
280 IF A=Ø THEN AA=1
290 IF A=128 THEN AA=2:POKE
    54273,9:FORZ=1TO2Ø:NEXT:
    POKE54273,Ø
300 IF AA>2 THEN AA=1
310 W=Ø:IF AA=1 THEN T=1:TT=Ø
320 IF AA=2 THEN T=Ø:TT=1:W=1
330 A1=A−W:FOR Z=1 TO 8:
    IFA(Z)<=A1 THEN B(Z)=T:C(Z)=TT:
    A1=A1−A(Z):GOTO35Ø
340 B(Z)=TT:C(Z)=T
350 IF B(Z)=1THENS=S+A(Z)
360 NEXT Z:A2=Ø:FORZ=1TO8:
    IFC(Z)=Ø THEN A2=A2+A(Z)
370 NEXT Z:A1=(255−A2)+T:IF
    A1>255 THENA1=Ø
380 FORZ=1TO 8:IFA(Z)<=A1+W THEN
    D(Z)=1:A1=A1−A(Z):SS=SS+A(Z):
    GOTO4ØØ
390 D(Z)=Ø
400 NEXT Z
410 PRINT'▀▄▄▄▄";MID$
    (A$,AA,1)"□□□□▌▌▌▌▌▌▌"A
420 PRINT"○"V$;:FORZ=1TO8:PRINT
    B(Z);:NEXT Z
430 ZA=INT(S/16):ZB=S−(16*ZA):
    PRINT"□□□□□"MID$(Z$,ZA+1,1);
    MID$(Z$,ZB+1,1)
440 PRINT"▄▄▄▄"V$;:FORZ=1TO8:
    PRINTC(Z);:NEXT:PRINT
450 PRINT"▄▄▄▄▄▄▄";:
    IF AA=1THENPRINT"−";
460 IF AA=2THENPRINT"+";
470 PRINT"□□□□▌▌▌▌▌▌▌"A:
    PRINT"○"V$;
480 FORZ=1TO8:PRINTD(Z);:NEXT Z:
    ZA=INT(SS/16):ZB=SS−(16*ZA)
490 PRINT"□□□□□"MID$(Z$,ZA+
    1,1);MID$(Z$,ZB+1,1): GOTO15Ø
500 I$="":PRINT"⌂▀▄▄▄
    ▄▄▄▄▄▄▄▄▄▄▄▄▄
    ▄▄▄▄▄ INPUT NUMBER?
    (−128 TO +127):□□□□
    □□▌▌▌▌▌▌▌▌";
510 FOR Z=1 to 4
520 GETJ$:IFZ=1 AND (J$="−" OR
    J$="+")THEN U$=J$:PRINTU$;:
    NEXT Z
530 IF Z=1 THEN 52Ø
540 PRINT "*▌▌□▌▌";:IF J$=""
    THEN 52Ø
550 IF J$=CHR$(13) THEN 61Ø
560 IF J$=CHR$(2Ø) THEN 5ØØ
570 IF ASC(J$)<48 OR ASC(J$)>57
    THEN 52Ø
580 I$=I$+J$:PRINT J$;:NEXT Z
590 GET J$:IF J$=CHR$(2Ø) THEN 5ØØ
600 IF J$<>CHR$(13) THEN 59Ø
610 IF VAL(I$)<Ø OR (VAL(I$)>
    128ANDU$="−") OR (VAL(I$)>127
    ANDU$="+") THEN5ØØ
620 IF U$="−"THEN AA=2:GOTO64Ø
630 AA=1
640 PRINT:PRINT"○□□□□□□□
    □□□□□□□□□□□□□□
    □□□□□□□□□□□□□□
    □□□□";:A=VAL(I$):GOTO28Ø
```

```
20 POKE 36878,15
30 FOR Z=1 TO 8:READ A(Z): NEXT Z:
   DATA 128,64,32,16,8,4,2,1
40 Z$="Ø123456789ABCDEF":A=Ø:A$=
   "+−":AA=1:POKE 65Ø,128:V$=
   "▙▙▙"
50 POKE 36879,25:PRINT
   "▙■▩▩▩▩◥◥
60 FOR Z=1 TO 9:PRINT "▙▙◪
   □□□□□□□□□□□
   □□□□□□":NEXT Z
70 PRINT "▩⊞▙▙▙▙2'S
   COMPLIMENT"
80 PRINT "▣▙▙▙NEGATIVE
   NUMBERS"
90 PRINT "▙▙▙−−−−−−
   −−−−−−−−−−"
100 PRINT "▓DEC□□□□□□
   BIN□□□□□□HEX"
110 PRINT "⊞▩▩▩▩▙▙▙
   FLIP BITS":PRINT TAB(1Ø4)"+1"
120 PRINT "◙◙◙◙▙+"TAB(19)
   "+"
130 PRINT "▩▩▩▩▩▩▩▩
   −−−−−−−−−−−−
   −−−−−−−−−−−";
140 PRINT "Ø□Ø□Ø□Ø□Ø□Ø□
   Ø□Ø□Ø□ØØ":GOTO 3ØØ
150 GET K$:IF K$="" THEN 15Ø
160 S=Ø:SS=Ø:IF K$<>"□" AND
   K$<>"B" THEN 5ØØ
170 A1=A:IF K$<>"□" THEN 21Ø
180 IF AA=2 THEN A1=A1−1
190 IF AA=1 THEN A1=A1+1
200 GOTO 24Ø
210 IF K$<>"B" THEN 24Ø
220 IF AA=2 THEN A1=A1+1
230 IF AA=1 THEN A1=A1−1
240 IF A1<Ø OR A1>128 THEN
   AA=AA+1
250 IF A1>128 THEN A1=127
260 IF A1<Ø THEN A1=1
270 A=A1
280 IF A=Ø THEN AA=1
290 IF A=128 THEN AA=2:POKE
   36876,2ØØ:FOR Z=1 TO 2Ø:NEXT Z:
   POKE 36876,Ø
300 IF AA>2 THEN AA=1
310 W=Ø:IF AA=1 THEN T=1:
   TT=Ø
320 IF AA=2 THEN T=Ø:TT=1:
   W=1
330 A1=A−W:FOR Z=1 TO 8:IF A(Z)
   <=A1 THEN B(Z)=T:C(Z)=TT:
   A1=A1−A(Z):GOTO 35Ø
340 B(Z)=TT:C(Z)=T
350 IF B(Z)=1 THEN S=S+A(Z)
360 NEXT Z:A2=Ø:FOR Z=1 TO 8:IF
   C(Z)=Ø THEN A2=A2+A(Z)
370 NEXT Z:A1=(255−A2)+T:IF A1>255
   THEN A1=Ø
380 FOR Z=1TO8:IFA(Z)<=A1+W THEN
   D(Z)=1:A1=A1−A(Z):SS=SS+A(Z):
   GOTO4ØØ
390 D(Z)=Ø
400 NEXT Z
410 PRINT "▣⊞▩▩▩▩◪"
   MID$(A$,AA,1):PRINT "▣▩▩▩
   □□□□▌▌▌▌◪"RIGHT$
   (STR$(A),LEN(STR$(A))−1)
420 PRINT "▣▩▩▩"V$;:FOR Z=1
   TO 8:PRINT RIGHT$(STR$(B(Z)),1)
   "□";:NEXT Z
430 ZA=INT(S/16):ZB=S−(16*ZA):
   PRINT "◪"MID$(Z$,ZA+1,1)
   MID$(Z$,ZB+1,1)
440 PRINT "▩▩▩▩▩"V$;:FOR
   Z=1 TO 8:PRINT RIGHT$(STR$(C(Z)),1)
   "▙";:NEXT Z:PRINT
450 PRINT "▩▩▩▩▩▩◪";:
   IF AA=1 THEN PRINT "−"
460 IF AA=2 THEN PRINT "+"
470 PRINT "□□□□▌▌▌▌◪"
   RIGHT$(STR$(A),LEN(STR$(A))−1):
   PRINT "◙"V$;
480 FOR Z=1 TO 8:PRINT RIGHT$
   (STR$(D(Z)),1)"□";:NEXT Z:
   ZA=INT(SS/16):ZB=SS−(16*ZA)
490 PRINT "◪"MID$(Z$,ZA+1,1)
   MID$(Z$,ZB+1,1):GOTO 15Ø
500 I$="":PRINT "▲▣▩▩▩▩
   ▩▩▩▩▩▩▩▩▩▩▩
   ▩▩▩▩INPUT NUM?
   ▓(−128▲TO▓+127)>□□□□
   ▌▌▌▌";
510 FOR Z=1 TO 4
520 GET J$:IF Z=1 AND (J$="−" OR
   J$="+")THEN U$=J$:PRINT U$;:
   NEXT Z
540 PRINT "*▌□▌";:IF J$="" OR Z=1
   THEN 52Ø
550 IF J$=CHR$(13) THEN 61Ø
560 IF J$=CHR$(2Ø) THEN 5ØØ
570 IF ASC(J$)<48 OR ASC(J$)>57
   THEN 52Ø
580 I$=I$+J$:PRINTJ$;:NEXT Z
590 GET J$:IF J$=CHR$(2Ø)
   THEN 5ØØ
600 IF J$<>CHR$(13)
   THEN 59Ø
610 IF VAL(I$)<Ø OR(VAL(I$)>128
   AND U$="−")OR(VAL(I$)>127
   AND U$="+") THEN 5ØØ
620 IF U$="−"THEN AA=2:
   GOTO 64Ø
630 AA=1
640 PRINT:PRINT "◙◙□□□□
   □□□□□□□□□□□□□□□□
   □□□□□□□□□□□□";:A=VAL(I$):
   GOTO28Ø
```

```
10 MODE 1
20 VDU 23;82Ø2;Ø;Ø;Ø;
30 VDU 19,1,6,Ø,Ø,Ø,Ø,Ø
40 PRINTTAB(13,3)"NEGATIVE NUMBERS"
50 PRINTTAB(13,4)STRING$(16,CHR$
   (224))TAB(6)"Dec"TAB(19)"Bin"
   TAB(31)"Hex"
60 PRINTTAB(8,13)"+"TAB(33,13)"+"
70 PRINTTAB(11,18)"2's Complement"
80 PRINTTAB(8,22)"Ø□□□Ø□Ø□Ø
   □Ø□□Ø□Ø□Ø□Ø□□
   □□ØØ"
90 PRINTTAB(7,27)"PRESS SPACE BAR
   TO INCREMENT"
100 PRINTTAB(7,29)"PRESS LETTER B
   TO DECREMENT"
110 GCOLØ,1
120 MOVE32Ø,48Ø:MOVE32Ø,7Ø4:PLOT85,
   96Ø,48Ø:PLOT85,96Ø,7Ø4
130 PRINTTAB(12,11)"FLIP BITS"
140 PRINTTAB(26,15)"+1"
150 VDU 31,6,21,224,224,224,31,31,
   21,224,224,224
160 PRINTTAB(11,21)STRING$(18,
   CHR$(224))
170 ?&7Ø=Ø
180 T=?&7Ø:IF T=128 THEN SOUND1,
   −15,1ØØ,1Ø
190 PROCBIN(8): PROCHEX(8)
200 T=T+256*(T>127):PROCDEC(8)
210 T=255−T:PROCBIN(13)
220 T=T+1:PROCBIN(2Ø):PROCHEX(2Ø)
230 T=T+256*(T>127):PROCDEC(2Ø)
240 *FX21,Ø
250 G=GET
260 IF G=32 THEN?&7Ø=?&7Ø+1:
   GOTO 18Ø
270 IF G=66 THEN?&7Ø=?&7Ø−1:
   GOTO 18Ø
280 PRINTTAB(Ø,3Ø);:INPUT?&7Ø:
   PRINTTAB(Ø,3Ø)STRING$(39,"□");:
   GOTO18Ø
290 DEF PROCBIN(Y)
300 FOR X=Ø TO 7
310 IF −(T AND 2∧X) THEN PRINT
   TAB(27−X*2+(X>3),Y)"1" ELSE
   PRINT TAB(27−X*2+(X>3),Y)"Ø"
320 NEXT X
330 ENDPROC
340 DEF PROCHEX(Y)
350 X=(T AND 24Ø)/16
360 A$=CHR$(X+48−7*(X>9))
370 X=(T AND 15)
380 B$=CHR$(X+48−7*(X>9))
390 PRINTTAB(32,Y);A$+B$
400 ENDPROC
410 DEF PROCDEC(Y)
420 PRINT TAB(6−LEN(STR$(T)),Y);
   "□□□"T:ENDPROC
```



In the sign convention, 128 acts as its own negative, and the computer starts counting upwards again

```
10 CLS
20 PRINT@8,“NEGATIVE NUMBERS”;
30 PRINT@40,STRING$(16,
   CHR$(131));
40 PRINT@65,“DEC”TAB(15)“BIN”
   TAB(28)“HEX”
50 PRINT@226,“ + ”TAB(29)“ + ”;
60 PRINT@361,“2’S COMPLEMENT”
70 PRINT@483,“Ø□□□□Ø□Ø
   □Ø□Ø□□Ø□Ø□Ø□Ø□□
   □□□ØØ”;
80 FOR J=1474 TO 15Ø2:POKE
   J,131:NEXT
90 FOR J=1 TO 7
100 FOR K=1 TO 24
110 POKE 1123+K+32*J,175
120 NEXT K,J
130 PRINT@165,“FLIP BITS”;
140 PRINT@313,“ +1”;
150 AT=AT AND 255
160 T=AT:IF T=128 THEN SOUND 3Ø,2
170 LN=3:GOSUB28Ø:GOSUB31Ø
180 T=T+256*(T>127):GOSUB34Ø
190 T=255-T:LN=7:GOSUB28Ø
200 T=T+1:T=T AND 255:LN=13:
   GOSUB28Ø:GOSUB31Ø
210 T=T+256*(T>128):GOSUB34Ø
220 IN$=INKEY$:IFIN$<>“B” AND
   IN$<>“□” AND IN$<>CHR$(13)
   THEN 22Ø
230 IF IN$=“B” THENAT=AT-1:
   GOTO15Ø
240 IF IN$=“□” THEN AT=AT+1:
   GOTO15Ø
250 PRINT@384,;:INPUT AT
260 PRINT@384,“□□□□□”;
270 GOTO 15Ø
280 FOR X=Ø to 7
290 IF-(T AND 2 ✝ X) THENPRINT
   @LN*32+23-X*2+(X>3),“1”; ELSE
   PRINT@LN*32+23-X*2+(X>3),“Ø”;
300 NEXT:RETURN
310 IF T<16 THENA$=“Ø” ELSE
   A$=“”
320 PRINT@LN*32+29,A$+HEX$(T);
330 RETURN
340 PRINT@32*LN,MID$(“□□□”+
   STR$(T),LEN(STR$(T)));
350 RETURN
```

SIGN CONVENTION

For most purposes, as mentioned above, a binary or hex number can act perfectly well both as a positive and a negative number. But sometimes you need to know whether a number is positive or negative.

When you want a program jump in machine code you have to specify by how many bytes you want the computer to jump with a positive number, to jump forwards; a negative number for backwards.

What the computer does is to look at the first bit of the binary number and decide for itself whether the number is negative or positive. If the first bit is 1, the computer takes it as negative. If it is Ø, the computer takes it as positive.

This process is known as the *sign convention*. It means that, instead of taking the eight-bit binary numbers to mean Ø to 255, the computer treats them as -128 to +127.

In the 2’s complement conversion program, you will notice that the computer beeps when you get to 128. This is because 128, which is 1ØØØØØØØ in binary, or 8Ø in hex, acts as its own negative. (You can check for yourself:—1ØØØØØØØ + 1ØØØØØØØ = (1)ØØØØØØØØ or Ø in eight-bit binary. Likewise 8Ø + 8Ø = (1)ØØ or Ø in two digit hex. And 128 - 128 = Ø in decimal.)

So which is it to be? As 1ØØØØØØØ has a 1 in its first bit, the computer treats it as a negative number: -128. On the other hand zero—or ØØØØØØØØ—has a Ø in its first bit. So the computer treats it as positive.

REFINING YOUR SCREEN GRAPHICS

Home computers offer plenty of scope to the budding artist. Here are some ways to expand your use of the BASIC graphics commands and create new screen pictures.

Once you have mastered the basics of drawing images on the screen, you can start to extend your artistic efforts by using some of the specialized graphics commands available on your computer. Commands like MOVE, PLOT, DRAW, PAINT and CIRCLE allow free rein to your imagination and allow you to create anything from a display chart to brighten up a business program, to the background for an exciting adventure game.

The article on page 84 showed how to use simple drawing commands to create line pictures on the screen and, in some cases, how to add colour. But you can extend these basic techniques with new ways of plotting dots, drawing lines, triangles, squares and circles. Used by themselves or in combination with colours, they become a powerful means of forming static or even moving images.

The way in which different computers use these commands varies from one to another, but each is capable of some really interesting effects. The exceptions are the Commodore 64 and Vic 20, whose standard BASIC does not include these commands. However, you can gain them by the addition of a suitable ROM cartridge (see page 87), and this article contains a section explaining how you can use this facility to extend your Commodore's drawing potential.

CIRCLES AND ARCS

Circles and arcs are among the most useful Spectrum 'tools' for drawing on-screen static graphics.

This golf course program shows how to use them to draw trees, fences, water, rough and bunkers, as well as describing the principles behind them.

You can check your progress as you go by RUNning each group of lines. Do not NEW your machine each time; if you leave the lines intact, you will end up with the scene in **fig. 1.**

Before you start on the circles, however, it is best to get the clubhouse out of the way. So start by entering these lines:

```
9Ø BORDER 4: PAPER 4:CLS
2ØØ LET w=1Ø: LET s=5Ø
21Ø FOR c=162 TO 174
22Ø PLOT INK 2;w,c
23Ø DRAW INK 2;s,Ø
24Ø LET w=w+2: LET s=s-4
25Ø NEXT c
26Ø FOR b=148 TO 162
27Ø PLOT INK 2;1Ø,b
28Ø DRAW INK 2;5Ø,Ø
29Ø NEXT b
295 DRAW INK 2;1Ø,-3: DRAW Ø,-11
3ØØ PRINT INK Ø;AT 2,2; "■"; AT 2,4;
   "■"; AT 2,6; "■"
```

Lines 2ØØ to 25Ø DRAW the roof, using techniques similar to those in the earlier chapter on plotting and drawing (pages 84 to 86). It starts off at 1Ø, 162 on the screen with a line which is 50 pixels wide. Then its width decreases by four pixels for every one pixel that it rises.

A similar loop in Lines 26Ø to 29Ø DRAWs the walls, with two extra lines (in Line 295) to form a porch. Then Line 3ØØ fills in the windows by the simplest possible method—PRINTing a black square from the standard ROM graphics characters through the walls at three places.

DRAWING AN ARC

On the Spectrum, as explained earlier (page 86) the easiest way to draw a complete circle is to use the CIRCLE command.

But it you want only part of a circle, the easiest way is to use an addition to a conventional DRAW statement. You can get a huge variety of effects by varying this statement, so it is worthwhile conducting a few experiments before you go any further. For example, try:

```
1Ø PLOT 13Ø, 3Ø
2Ø DRAW Ø, 1Ø, 1
3Ø GOTO 2Ø
```

(Don't worry about the error report.)

As you'll recall, the first two numbers in Line 2Ø tell the computer to DRAW a line from the PLOTted point to a spot 10 pixels above it. What the last number is doing is to make that line curve, rather than going in a straight line. How much of a curve you get is dictated by the size of the number. This tells the Spectrum to DRAW part of a circle. A whole circle is represented by 2 times π so the number 1 makes it DRAW about one-sixth of a circle (or, more accurately, 1 divided by 2 times π). If you try changing Line 2Ø to:

```
2Ø DRAW Ø, 1Ø, 2
```

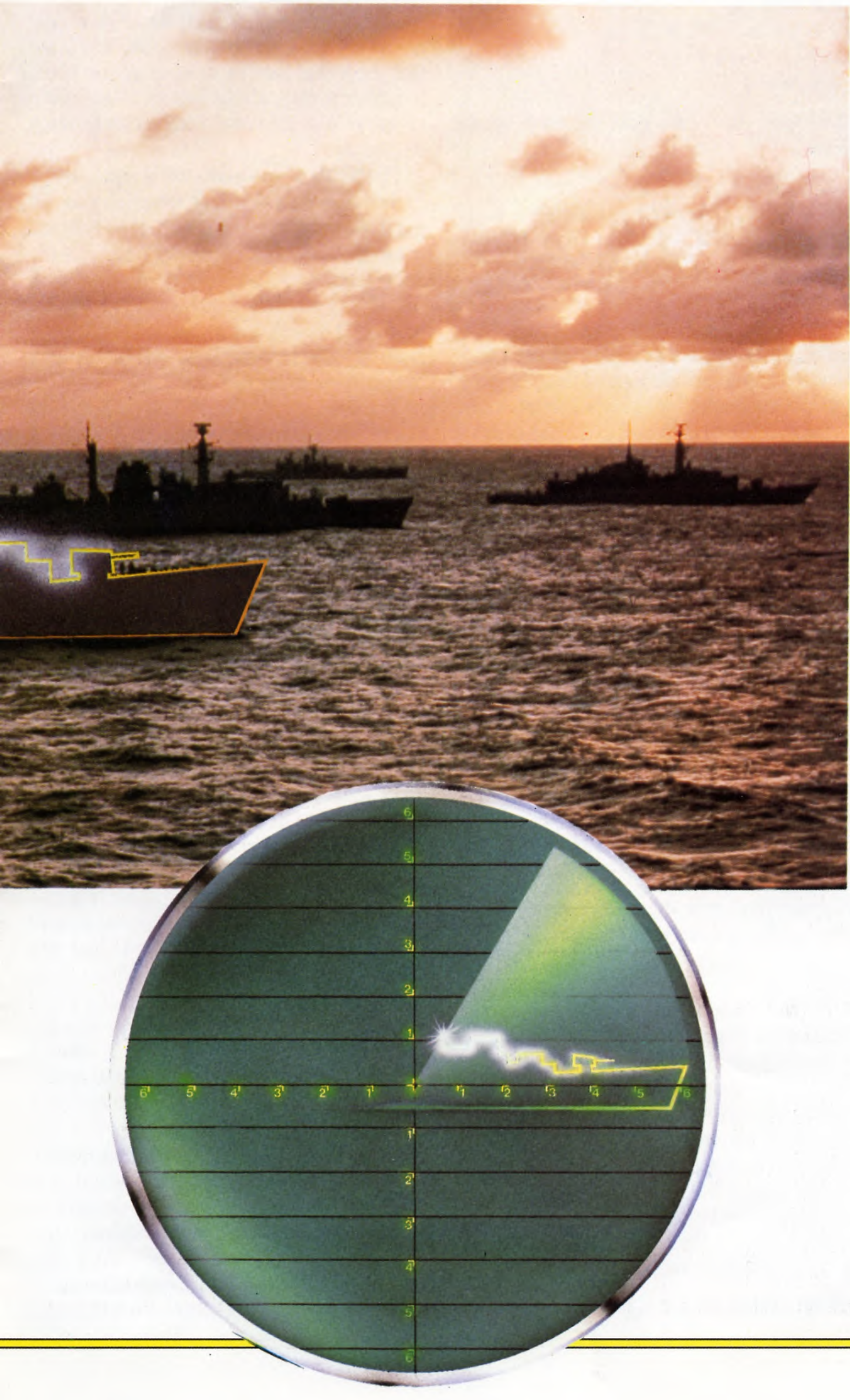
... you'll find you get a more pronounced series of curves. Similarly, Ø, 1Ø, 3 gives a sawtooth pattern, Ø, 1Ø, 4 gives part of a chain-link fence (or half the trunk of a palm tree, depending on how you view it!), while Ø, 1Ø, 6 produces a spiral.

If, however, you try

```
2Ø DRAW Ø, 1Ø, 2*PI
```

... you may get a surprise. What the Spectrum wants to do is DRAW a circle two of

- USING MOVING LINES ON THE DRAGON AND ACORN
- GOLF COURSE SCENE FOR SPECTRUM AND ACORN
- LETTERS ON THE DRAGON
- MORE WAYS WITH COLOUR
- PICTURES FROM CIRCLES AND ARCS
- SIMON'S BASIC FOR COMMODORE GRAPHICS



whose points are in a straight line. But since such a circle would be rather bigger than your living room—indeed, rather bigger than the solar system—it DRAWS only that part of it that it can.

For drawing scenery, try this:

```
10 PLOT Ø, 100
20 DRAW RND*5+5, Ø, 2
30 GOTO 20
```

This DRAWs the wave pattern of a choppy sea. For calmer waters, try RND*10+10 or even RND*15+10.

One point to remember about such arcs is that a *negative* number at the end of your DRAW line will produce the arc's mirror image. Now clear Lines 1Ø to 3Ø and you can continue building up the golf course scene.

FENCE AND LAKE

The golf course program has two examples of arcs used for graphic effect—the first for the little fence in front of the clubhouse; the second for the lake.

First type in these lines:

```
31Ø FOR f=Ø TO 84 STEP 3
32Ø PLOT f,142
33Ø DRAW 3,Ø,-3
34Ø NEXT f
345 DRAW 35,-42: DRAW -12,-6
```

Here the starting off point is Ø, 142 on the screen. What Line 33Ø does is to DRAW a series of tight arcs—semi-circles only three pixels wide—to form the fence. The number you get is governed by the FOR . . . NEXT loop.

Next, enter and RUN these lines:

```
1ØØ LET x=13Ø: LET y=125: LET z=5Ø
11Ø PLOT INK 5;x,Ø
12Ø DRAW INK 5;y,z,-1.25
13Ø LET x=x+1: LET y=y-1: LET z=z-1
14Ø IF x>254 THEN GOTO 17Ø
15Ø IF z<1 THEN LET z=Ø
16Ø GOTO 11Ø
```

The picture here is more complicated. First the computer PLOTs a point 130 pixels from the left and 0 pixels from the bottom of the screen. Then it DRAWs a line to a point 125 pixels to the right, and up 50 pixels from the bottom of the screen, 'bending' the line by −1.25 radians as it goes.

From that stage the variables take over. Variable x starts each line one pixel farther to the right, y makes the line one pixel shorter than the previous one (else it would run off the screen), while z makes the line end one pixel lower than the previous line.

Eventually, of course, z would become a minus number, making the computer try (but fail) to PRINT off the bottom of the screen. Hence the need for Line 15Ø, which makes all the short lines towards the end of the program finish on the bottom line of the screen.

TREES AND BUSHES

The golf course program also uses complete circles—as a substitute for a PAINT or similar statement, available on some computers but not on the Spectrum—to produce trees and scrub. These few lines give some bushes randomly behind the 'green':

```
400 FOR r=172 TO 168 STEP −1
410 LET x=RND*45+195
415 PLOT x,r−2: DRAW Ø,−2
420 CIRCLE x,r,RND*2+1
440 CIRCLE x+1Ø,r,RND*2+1
450 NEXT r
```

While these few give similar scrub on the right-hand side:

```
460 FOR r=135 TO 172 STEP 6
470 LET y=252
480 CIRCLE y,r,RND+2
490 NEXT r
```

The trees in the left-hand bottom corner are too big to be DRAWn randomly. So a different technique, READ ... DATA (see pages 104 to 109) is used instead:

```
900 FOR w=1 TO 3
910 READ a,b
920 PLOT a,b
930 DRAW Ø,−24
940 LET f=RND*5+5
950 CIRCLE a−1Ø,b+f,f: CIRCLE a,b+f,
   f: CIRCLE a+1Ø,b+f,f
960 CIRCLE a−5,b+f*2,f: CIRCLE a+5,
   b+f*2,f
970 CIRCLE a,b+f*3,f
980 NEXT w
3000 DATA 2Ø,7Ø,52,85,84,1ØØ
```

The trick here is to DRAW the trunks first, working downwards from the original PLOTting points, so that you do not get ugly trunk marks showing through the 'foliage'. The trunks are PLOTted at 2Ø, 7Ø and so on by the DATA in Line 3ØØØ. Line 94Ø randomizes the sizes of the foliage, while the b+f in Line 95Ø makes sure that the bottom rows of circles begin a suitable distance above the trunks.

FINISHING TOUCHES

The driving tee (not exactly pukka; more the rubber mat you might find on a municipal course!) is DRAWn by these lines:

```
1000 LET t=30
1010 FOR y=Ø TO 1Ø
1020 PLOT t,y
1030 DRAW −3Ø,3Ø
1040 LET t=t+2
1050 NEXT y
```

And these few lines give the flags on the greens:

```
170 PLOT 22Ø,14Ø
180 DRAW Ø,15: DRAW 8,−3: DRAW −8,−2
190 PLOT 22, 12Ø
195 DRAW Ø,18: DRAW 9,−3: DRAW −9,−2
```

Finally, you'll need some bunkers. The neatest—but a very slow—way of DRAWing these, in the absence of a PAINT statement, is to start with a tiny ellipse and make it grow, pixel by pixel, until it reaches a suitable size.

How to DRAW ellipses is explained in detail in a later article on the computer's mathematical functions. In the meantime, enter these lines, then go off and have a coffee while they are RUNning:

```
1495 LET r=1
1500 FOR x=Ø TO 2*PI STEP PI/18Ø
1510 PLOT INK 6;168+r*SIN
   x,147+r*COS x/2.5
1520 PLOT INK 6;235+r*SIN
   x,1Ø6+r*COS x/2.75
1530 PLOT INK 6;225+r*SIN
   x,97+r*COS x/2.5
1540 NEXT x
1550 LET r=r+2
1560 IF r>2Ø THEN GOTO 6ØØØ
1570 GOTO 15ØØ
```

Alternatively, you might like to make up simpler, but cruder, bunkers using UDGs!

With the addition of the Simons' BASIC cartridge a large number of additional programming commands are made available on the Commodore 64. Some of these are provided to simplify graphics programming and the use of several of these has already been explained (see pages 87–88).

ARC and ANGL are two extra drawing commands not yet covered. The first is used to draw parts of the circumference of a circle and the command takes the form:

```
99 ARC 15Ø,5Ø,6Ø,27Ø,1,3Ø,3Ø,1
```

This would normally be preceded by the HIRES command, so enter this line also:

```
90 HIRES Ø,1
```

The first pair of figures after the ARC command define the screen coordinates of the centre of the circle whose arc you require. As usual, these values refer to the pixel positions in the standard configuration of horizontal (X value) first, vertical (Y value) next.

Try adjusting one or both these figures and reRUN the program several times. You'll see that the higher each of these values becomes, the closer to the right or bottom of the screen the curve is plotted until you eventually reach the border.

The actual length of the arc is regulated by the next two pairs of values. The first pair (6Ø and 27Ø) are the start and end *angles*. If you imagine you're dealing with a clockface, the angle count begins and ends at 12 o'clock with the values Ø and 36Ø. Thus our example curve begins at 2 o'clock and ends at 9 o'clock. Once again, try changing the values of just this pair of figures to gauge the effect.

The next figure (1) is the *plotting increment* and by adjusting this to a value in the range 1 to 36Ø you choose the interval in degrees the curve produced using the value 1Ø will look somewhat coarser than that of the original program line. As an indication of the possibilities, make these amendments to the original program:

```
90 HIRES Ø,1: FOR N=1Ø TO 36Ø STEP 1Ø
99 ARC 16Ø,8Ø,Ø,36Ø,N,84,6Ø,1
100 NEXT : PAUSE 1Ø
```

This shows the effect of adjusting the plotting increment in steps of 1Ø degrees for a curve which starts and ends at the same point—in other words, a circle. (The use of variable, N, in this example suggests a powerful method of ringing the changes with minimal extra programming.)

The pair of values which follows N in this example regulates the physical appearance of the 'circle' whose arc is displayed. The first value is the length, in pixels, of the X or horizontal radius; the second is that for the Y or vertical radius. If you want a circular arc the X value must be 1.4 times the Y value in HIRES mode, and 1.6 times in MULTI mode. Otherwise an elliptical curve instead of a proper circle is the result.

Try adjusting these two values but remember that you have a maximum resolution of 320 pixels horizontally, 200 vertically in HIRES mode, and half the horizontal resolution in MULTI mode. The sum of the very first value after the ARC command—the X coordinate of the curve centre—and the value of the X radius of the curve must not exceed the horizontal limit. Similarly, the sum of the a problem: as you have seen, a byte can exceed 200. In both cases, exceeding the limit causes the curve to run along the screen edge until the curve reaches an allowable value. At this point it continues on its way again.

The final figure in the command line is the *plot type* value. Putting a Ø here would clear a dot, obviously of no use. Putting a 1 plots a dot, and a 2 inverses a dot (turns 'off' a dot that is 'on', and 'on' a dot that is 'off'). Substitute the value 2 in the program above to see what happens.

DRAWING RADII

The next command, ANGL, is used for drawing the radius of a circle and can be used in a variety of ways to depict things like the spokes of a wheel or the blades of a fan. It takes the form (NEW the computer):

```
1Ø HIRES Ø,1: FOR N=Ø TO 36Ø STEP 4
2Ø ANGL 16Ø,8Ø,N,84,6Ø,1
3Ø NEXT: PAUSE 1Ø
```

Again, to make the program do a lot of work with little programming, a variable is included as part of a FOR ... NEXT loop. The first pair of values after the ANGL command are the X and Y coordinates of the centre of the circle whose radius is to be drawn—the starting point of the radius, of course. The value N is the *angle* to the perpendicular at which the radius is to be drawn. Thus the value 45 here would draw a line from the start point to another at 3 o'clock to it. As you can see, values of Ø to 36Ø can be used. Try changing the STEP values from 1 to 1Ø, or setting a fixed value for N in Line 2Ø.

The remaining three values serve exactly the same function they do with the ARC command. But you can usefully look at the 'unplot' value Ø. Add these lines to your existing program:

```
3Ø NEXT
35 CIRCLE 16Ø,8Ø,84,6Ø,1
4Ø FOR N=36Ø TO Ø STEP -4
45 ANGL 16Ø,8Ø,N,84,6Ø,Ø
5Ø NEXT: PAUSE 1Ø
```

This repeats the pattern, adds a circle and Line 45 then proceeds to erase the work of Line 2Ø.

BLOCKS OF COLOUR

The command BLOCK enables you to create coloured rectangles. You could do this by using REC and PAINT but BLOCK is often more convenient, particularly where several rectangles are required. The command takes the form that follows (NEW your computer first):

```
1Ø HIRES Ø,1: MULTI 2,5,6
2Ø BLOCK 1Ø,1Ø,3Ø,3Ø,1
5Ø PAUSE 1Ø
```

When RUN, this displays a single colour rectangle. The first pair of values define the

top left corner of the rectangle, the next pair the bottom right corner. The final figure is the plot type, which, in MULTI mode (selected in the previous line), selects the first colour option—value 2, or red. Try changing the plot type value to 2 or 3 and reRUN the program. Better still, add these lines:

```
3Ø BLOCK 2Ø,2Ø,5Ø,5Ø,2
4Ø BLOCK 3Ø,3Ø,7Ø,7Ø,3
```

And three colour blocks should result when you RUN the program! For a more interesting result make the following amendments which display a random sequence of fifteen red, green and blue blocks:

```
2Ø N=5: M=1Ø: C=1
3Ø BLOCK N,N,M,M,C
4Ø N=N+5: M=M+1Ø:
   C=INT(RND(1)*3+1)
5Ø IF M<16Ø THEN GOTO 3Ø
6Ø PAUSE 1Ø
```

DRAWING

DRAW and ROT are two commands used together to design and display a specified shape. A sequence of numbered instructions are used in the form below (NEW the computer before typing in):

```
1ØØ DRAW A$,15Ø,15Ø,1
```

The A$ component of this instruction actually defines the shape of the drawing, the next pair of values are the X and Y coordinates of the starting point, and the final figure is the familiar plot type value.

A$ can contain numerals from Ø to 9 and these values signify the following:

Ø move one pixel right
1 move one pixel up
2 move one pixel down
3 move one pixel left
4 invalid
5 move right and plot pixel
6 move up and plot pixel
7 move down and plot pixel
8 move left and plot pixel
9 stop drawing

Any combination of numbers may be used in the string (A$) with a maximum of 74 in any single line to allow room for the other parameters of the DRAW command. Elaborate drawings require more plotting instructions than may be contained in a single line but you can get round this by *concatenating* several strings to give a maximum string length of 255 characters. An example of this is:

```
A$="A STRING OF 74 CHARACTERS"
A$=A$+"EVEN MORE CHARACTERS"
```

Which greatly lengthens A$! Now let's look at an example of A$: (Enter this)

```
1Ø A$="55555555555555555555657
   5657555555555Ø6Ø163788888888
   888888888888888888"
2Ø A$=A$+"1ØØØØØØØØØØØØØØ555Ø
   6881555Ø6886551888836555188
   88836555550Ø63"
3Ø A$=A$+"888888888888888811
   Ø55555555557555550638888655 5
   55Ø68888888865"
4Ø A$=A$+"555555ØØØ688381Ø55
   551889"
```

This in fact defines a downhill skier! But additional programming is required to display A$ and this is where the command ROT comes into play.

ROT allows you to specify the orientation and size of the defined shape of the DRAW command. Add this line to the program:

```
2ØØ ROT Ø,1
```

The first parameter after the command specifies the degree of rotation of the drawing. The following values show the range of orientation that is possible:

Ø Ø degrees rotation
1 45 degrees rotation
2 9Ø degrees rotation
3 135 degrees rotation
4 18Ø degrees rotation
5 225 degrees rotation
6 27Ø degrees rotation
7 315 degrees rotation

You'll be able to try these as alternative values in Line 2ØØ when the program is eventually ready to RUN.

The second value after the ROT command specifies the size of the drawing. With the value 1, the shape is displayed at normal size. Increasing this figure enlarges the drawing but you must avoid 'overshooting' the available screen area. To get the program to RUN properly, enter these additional lines:

```
5Ø HIRES Ø, 1
25Ø PAUSE 2Ø
```

Try RUNing the program several times, altering values where possible. Strange things may happen, but always try reRUNing the program a second time if an anticipated effect fails to materialize.

Now add these extra lines to the existing program to see how a group of Simons' BASIC commands may be used:

```
5Ø HIRES Ø,1:MULTI 2,5,3:COLOUR 6,1
6Ø LINE Ø,2Ø,32Ø,6Ø,1
65 LINE 32,1,8Ø,7,1
7Ø LINE Ø,1Ø,32,1,1
8Ø LINE 25,25,1ØØ,1,1
85 LINE 1ØØ,1,25Ø,5Ø,1
9Ø CIRCLE 15Ø,2Ø,5,6,1
95 PAINT 5,5,3
115 FOR N=1 TO 5:READ X1,X2,X3,X4,X5
12Ø FOR Z=1 TO X1 STEP 4
125 LINE X4,X5+Z,X3,(X5-2Ø)+Z,
   2+RND(1)*2
13Ø LINE X2,X5+Z,X3,(X5-2Ø)+Z,
   2+RND(1)*2:NEXT
14Ø BLOCK X3,X5+X1-2Ø,X3,
   X5+X1,1::NEXT
   N:ROT Ø,1
2ØØ LOW COL 1,4,2:FOR Z=25 TO 6ØSTEP 3:
   DRAW A$,Z-3,163,1
2Ø5 DRAW A$,Z,163,2:NEXT Z
21Ø DRAW A$,25,169,3
1ØØØ DATA 5Ø,33,4Ø,47,9Ø
1ØØ2 DATA 2Ø,5,1Ø,15,3Ø
1ØØ4 DATA 25,15,2Ø,25,6Ø
1ØØ6 DATA 12Ø,1,1Ø,19,9Ø
1ØØ8 DATA 2Ø,85,9Ø,95,3Ø
```

Although the Vic 2Ø has excellent graphics you really need to add a Super Expander cartridge before you can make good use of them. The cartridge provides all the usual graphics commands such as DRAW, CIRCLE and so on, and the program below shows how you can create a quite complex picture using these simple commands:

```
1Ø GRAPHIC 2:COLOR 1,6,6,Ø
2Ø CIRCLE 2,5ØØ,5ØØ,1ØØ,4ØØ
25 FOR Z=1 TO 15Ø STEP 1Ø
3Ø CIRCLE 2,5ØØ+SIN(Z)*3Ø,
   5ØØ+Z,1ØØ,4ØØ,6Ø,13Ø:NEXT
35 FOR Z=1 TO 2Ø STEP 7
4Ø CIRCLE 2,45Ø,44Ø,3Ø-Z,2Ø
45 CIRCLE 2,55Ø,44Ø,3Ø-Z,2Ø
5Ø DRAW 2,47Ø,74Ø+Z TO 49Ø,
   72Ø+Z TO 51Ø,74Ø+Z TO 53Ø,
   72Ø+Z TO 55Ø,74Ø+Z:NEXT
6Ø DRAW 2,49Ø,5ØØ TO 47Ø,64Ø
   TO 49Ø,66Ø TO 51Ø,65Ø
7Ø CIRCLE 2,51Ø,75Ø,4Ø,2Ø,1Ø,4Ø
8Ø FOR Z=1 TO 5ØØ STEP2Ø
9Ø DRAW 2,4ØØ,52Ø TO 2ØØ-(Z*.3),
   6ØØ+Z TO 52Ø,1ØØØ TO 8ØØ+
   (Z*.3),6ØØ+Z TO 6ØØ,5ØØ:NEXT
1ØØ FOR Z=1 TO 2:CHAR 15-Z,8-Z,
   "*":NEXT
11Ø GOTO 11Ø
```

The program uses a series of ellipses, curves and straight lines to build up a picture of a woman's face. Lines 2Ø and 3Ø draw the face and hair, Lines 35 to 7Ø draw the eyes, nose and mouth, and Lines 8Ø to 1ØØ draw the shoulders. If you have a cartridge see if you can create a man's face in the same way.

PLOT and DRAW commands don't just limit you to simple straight line drawing of the type covered in detail on page 88. With the right controls, curves, zig-zags and a whole range of textures are at your disposal. And with new uses for colour as well, you have the basis for a whole range of visual effects.

The article on page 84 showed how you can get to grips with simple drawing using PLOT and DRAW—and how to add colour to your pictures with just a few more instructions. Once you've mastered the basic commands, you can add a new dimension to your graphics by introducing lines whose shape can be controlled as they are being drawn. You can add waves to the surface of your seas, put jagged rocks on the outline of a cliff face, or build battlements on a castle. All of these can be done with a single line.

The secret of adding texture with PLOT and DRAW commands is not to be limited by the fixed coordinates that these specify to draw a single line, but to use the computer's speed to draw 100 or more short lines, each at a different angle, across the screen. And instead of one zig-zag line, you can draw 50 or even 2000 at different heights across the screen to produce a wide range of different effects. For example, a program could move the cursor a random distance up and down for every unit that it moves to the right. This would draw an irregular jagged line—just what you need for drawing rocks, or perhaps a cracked pane of glass.

TROUBLE SHOOTER

Unlike most computers, the Acorn machines allow you to use the whole of the TV screen for your text or drawing. However, on some TV sets you might find that the display is too far up or too far down the screen so that you lose part of the picture. This commonly happens near the corners of the screen.

Luckily, the BBC computer has a way to correct this using the *TV command. *TV 1 moves the display up one line and *TV 255 moves it down one line. Other numbers between 1 and 255 move the display even more. Typing a ,1 after it turns off the interlace (fuzziness).

But note that these commands only come into effect after you press BREAK or change MODE.

DRAWING RANDOM LINES

One example of this type of control uses FOR ... NEXT loops to alter the co-ordinates of a line randomly between limits. To see how this works, enter the lines below and RUN the program. Do not be put off by the % signs; they are there just to make the plotting faster. The program would work without them.

```
1Ø MODE 2
2Ø GCOL Ø,1
3Ø LET Y%=8ØØ
4Ø FOR X%=Ø TO 1279
5Ø LET Y%=Y%-3Ø
6Ø IFY%<Ø THEN END
7Ø MOVE X%, Y%
8Ø FOR A%=1 TO 15Ø
9Ø PLOT 1,RND(2Ø),RND(3Ø)-15
1ØØ NEXT A%,X%
```

What this does is to give a series of lines across the screen. To achieve this, a colour mode (Line 1Ø) and a colour in which to draw (Line 2Ø) are selected.

The crux of the program is Line 9Ø, which draws a line from a point on the left edge of the screen to a random point to the right, then from a point near this new point to another random point, and so on across the screen. The starting point is set up at Lines 3Ø to 5Ø. These lines set X%=Ø and Y%=77Ø. Line 9Ø lets the cursor move to the first random point and PLOTs. The next starting point is selected at Lines 4Ø and 5Ø, and the next finishing point is selected at Line 9Ø.

The effect will vary according to the number of times each line is drawn and the limits you set for A% in Line 8Ø. The variations on this routine are numerous. Try changing each value in turn to see the result. You might want a perspective view, to give the effect of a mountain range—as you will see when the next block of program is RUN. It is almost the same as the one above, but with different values:

```
1Ø MODE 2
2Ø GCOL Ø,1
3Ø LET Y%=-18Ø
4Ø FOR X%=-1Ø TO 1279
5Ø LET Y%=Y%+2Ø
6Ø IF Y%>15Ø THEN END
7Ø MOVE X%,Y%
8Ø FOR A%=1 TO 15Ø
9Ø PLOT 1,RND(2Ø),RND(4Ø)-15
1ØØ NEXT A%,X%
```

DRAWING AN AREA OF ROUGH

Alternatively, you might want just a small patch of the screen textured by drawing lines

1. Use arcs and circles for the Spectrum golf course

2. Simons' BASIC makes drawing easy on the Commodore

across the television screen. The next block of program could represent a growth of tall grass or weeds, just the thing for an area of rough ground—in a game of golf, for example. Type NEW, then type in and RUN these next few lines:

```
1Ø MODE 2
2Ø GCOL Ø,13Ø:GCOL Ø,1
3Ø CLG
4Ø FOR T=1 TO 5Ø
5Ø MOVE 6ØØ+RND(1ØØ),6ØØ+RND(1ØØ)
6Ø FOR A=1 TO 1Ø
7Ø PLOT 1,RND(2Ø),RND(3Ø)-15
8Ø NEXT A,T
```

Line 3Ø clears the screen to green—the background colour selected at Line 2Ø. This time, though, the starting points are randomly selected (Line 5Ø) each time the program looks at Line 4Ø. The finishing points, too, are randomly selected and PLOTted (Line 7Ø) each time the program looks at Line 6Ø. This gives an irregular edge to the textured area, instead of a sharp rectangular edge. Again, the effect can be varied extensively. Try the program in MODE 1, then change Line 1Ø back to MODE 2.

MAKING A BUNKER

The next element in a golf course might be a sand obstacle or bunker. The technique of PLOTting random co-ordinates is again a useful solution to this problem. Add these lines to the program above:

```
9Ø GCOL Ø,3
1ØØ FOR T=Ø TO 15Ø
12Ø MOVE 2ØØ+RND(1ØØ),5Ø+RND(1ØØ)
13Ø FOR A=Ø TO 1Ø
14Ø PLOT 65,2Ø,RND(5Ø)-1Ø
15Ø NEXT A,T
```

This time the PLOT command displays not lines, but single points. As for the rough, a random starting position is set up (Line 12Ø). Each time the program steps through the FOR ... NEXT loop starting at Line 1ØØ, a single point is PLOTted 20 units away in the X direction and -9 to 40 units in the Y direction.

Here is another picturesque example of the dot shading techniques:

```
2ØØ MODE 2
21Ø GCOL Ø,13Ø
22Ø CLG
23Ø GCOL Ø,3
24Ø FOR Y%=1Ø24 TO 1ØØ STEP -8
25Ø FOR X%=Ø TO 1279 STEP RND(5)+1Ø
26Ø IF RND(12ØØ)<Y%THEN PLOT 69,X%,Y%
27Ø NEXT X%,Y%
```

Line 21Ø selects a green background, then Line 26Ø builds up a picture of shaded dots. The striking perspective of the whiter background is given by the IF ... THEN condition at Line 26Ø, which PLOTs points less frequently as the program advances.

This type of control is ideal when you wish to make a picture or set the scene for a game. And you can bring several elements together or reposition them simply for different views. For example, here are some lines to bring together the rough, bunker and two woods:

```
2ØØ GCOL Ø,Ø
21Ø FOR Y%=1Ø23 TO 85Ø STEP -1Ø
22Ø MOVE Ø,Y%
23Ø DRAW 15Ø,Y%
24Ø NEXT Y%
245 REM......WOODS 2
```

```
25Ø FOR Y%=84Ø TO 7ØØ STEP -1Ø
26Ø MOVE Ø,Y%
27Ø LET X%=Y%-69Ø
28Ø DRAW X%-2Ø,Y%
29Ø NEXT Y%
295 REM......WOODS 3
4ØØ LET X1%=5ØØ
41Ø LET X2%=11ØØ
42Ø FOR Y%=1Ø23 TO 825 STEP -1Ø
43Ø LET X1%=X1%+1Ø
44Ø MOVE X1%,Y%
45Ø LET X2%=X2%-2Ø
46Ø DRAW X2%,Y%
47Ø NEXT Y%
```

Notice that as you enter these lines they will overwrite the previous lines for the picture.

COMPLETING THE GOLF COURSE

Enter the next block of lines to change an area of the background colour to form a lake:

```
5ØØ GCOL Ø,4
51Ø MOVE 1279,65Ø
52Ø MOVE 1279,6ØØ
53Ø PLOT 85,11ØØ,6ØØ
54Ø PLOT 85,1ØØØ,4ØØ
55Ø MOVE 1279,6ØØ
56Ø PLOT 85,1279,1ØØ
```

This block of program PLOTs and fills three triangles—the first is set up at Lines 51Ø, 52Ø and 53Ø, the second at Lines 52Ø, 53Ø and 54Ø and the third at Lines 54Ø, 55Ø and 56Ø. Enter the next block and RUN the program to see a colourful course complete with a flag to mark one of the holes and a rectangle to mark a building:

```
59Ø REM......BUILDING
595 GCOL Ø,1
6ØØ MOVE Ø,5ØØ
61Ø MOVE 5Ø,5ØØ
62Ø PLOT 85,Ø,3ØØ
63Ø PLOT 85,5Ø,3ØØ
64Ø REM......FLAG
7ØØ MOVE 3ØØ,8ØØ
71Ø DRAW 3ØØ,9ØØ
72Ø GCOL Ø,6
73Ø MOVE 38Ø,88Ø
74Ø PLOT 85,3ØØ,86Ø
```

Both objects are PLOTted as triangles—in a similar way to before.

Building up the golf course in stages like this lets you test and refine each element before adding it to the whole picture. Now you know how it's done, see if you can design a different picture, like the snow scene that is illustrated on page 189.

THE DRAW COMMAND

The LINE and CIRCLE commands are very useful for drawing simple or regular shapes with little effort by the programmer. But as you try to draw more sophisticated shapes you'll find that your programs become very long and unwieldy. Imagine how long the program would be to draw something like the ship in fig. 1, if you had to use a LINE statement for each separate line on the ship.

To avoid this problem, you need to use the computer's DRAW command. Using DRAW, you can direct the course of a line while it is being drawn. You can tell the computer to take the line a set distance to the right, then some other distance up, then to the left, and so on. You give it these instructions as a string, so the program can be much more compact.

As an example, type in and RUN this program, which creates the ship in only ten lines:

```
1Ø PMODE 4,1
2Ø PCLS5
3Ø SCREEN1,1
4Ø DRAW"BM23,96CØ"
5Ø DRAW"R28E2U3L6UR6E2R5F2D3R3U2R7D
   2R4U9E2R4F2D5R3U2R4U6E3R5D8"
6Ø DRAW"R3U24L3UR8DL4D24RF2R5D4R4U4
   R6U9E3R5D1ØR4U2R4U14RD1ØR3U2"
7Ø DRAW"R4D11R7U3E2R4F2D3RD8R3U5E2R
   8D2R6U2R7UE2R6F2D4R4U4E2R5F2"
8Ø DRAW"R6DL6D3R24G12L195H4U5"
9Ø PAINT(127,1ØØ),Ø,Ø
1ØØ GOTO 1ØØ
```

The program works like this:

Lines 1Ø to 3Ø set up the initial conditions. Line 1Ø selects PMODE 4 so that the highest resolution can be used. PCLS 5 in Line 2Ø clears the screen and changes it to Buff.

The series of DRAW commands in Lines 4Ø to 8Ø form the outline of the ship. In each case, DRAW operates on the string of instructions, which are contained within the inverted commas which follow.

Line 4Ø is the simplest, with a short string that contains instructions telling the machine where to start DRAWing, and in what colour. The first instruction is the letters BM. This stands for Blank Move, and positions the 'drawing pen' at the place you want your graphic to start. The starting position is given by the co-ordinates which follow it, in this case 23,96—notice that these aren't enclosed by brackets as in some of the other graphics commands. The final instruction in the string is the DRAW colour, CØ or Black.

Line 5Ø starts to trace the outline. The string which controls the DRAW command may look very confusing, but it's really very simple. It consists of a series of directions and distances. The letters control the direction of the line and the numbers give its length in pixels. If there isn't a number after a direction, then it will move one pixel only.

There are eight directions which you can use: U means Up, D means Down, L means Left, R means Right, E means draw at 45 degrees (up to the right), F means draw at 135 degrees (down to the right), G means draw at 225 degrees (down to the left) and H means draw at 315 degrees (up to the left).

Reading from the start of Line 5Ø, the strings tells the computer to draw right 28 pixels, at 45 degrees 2 pixels, up 3 pixels, and so on. Try translating the string into pencil movements on graph paper to see how the ship's outline builds up.

Lines 6Ø to 8Ø contain similar strings which complete the ship's outline. You could use just one long string instead of all these lines, but a large number of instructions tends to get unwieldy and hard to correct. Finally, the outline is turned into a silhouette by Line 9Ø which PAINTs the ship black.

DRAWING LETTERS

One of the limitations with graphics programs is that the Dragon and Tandy can't display text on the graphics screen. This means that you can't PRINT out a score, for example, on a game that uses the high resolution graphics. The game on page 98, for example, is all displayed on the text screen for that reason.

There is a solution to this problem, though. You can design your own characters using the DRAW command. Type in and RUN this program and you'll see how HELLO can be DRAWn:

```
1Ø PMODE 3,1
2Ø PCLS
3Ø SCREEN1,Ø
4Ø HE$="D4BR3U2NL3U2BR5L3D2NR2D2
   R3BR5L3U4BR5D4R3BR2U4R3D4L3"
5Ø DRAW"BM11Ø,5Ø;C3S8"+HE$
6Ø GOTO 6Ø
```

The message is DRAWn in PMODE 3, a four-colour mode.

Line 4Ø shows you another characteristic of the DRAW command. As it operates on a string of instructions you can define string variables and call them up in a DRAW command later in the program.

The instructions in HE$ tell the computer how to DRAW the letters in HELLO. Try translating the instructions into pen movements as you did before, remembering that B means blank, and no line will appear from DRAWing with an instruction preceded by a B.

If you have several words you wish to display on the graphics screen, such as BAD LUCK, WELL DONE, and so on, you can set up further strings, BL$ and WD$ for example. Try working out the instructions for one of these on graph paper. Once you have defined the strings, you can call these up whenever they are needed in the program.

HELLO is DRAWn by Line 5Ø. The start position is at 11Ø,5Ø, the colour is number 3 (Blue) so you'll see a blue word, and in size 8 (double scale). S may be followed by a number from 1 to 62—1 is quarter size, 4 is normal size (the *default*, which you get in the absence of instructions) and 8 is double size.

As Line 5Ø demonstrates, you can join up strings of DRAW instructions just like any other strings. If you want to call up other strings—such as BL$ or WD$—you can set them up in just the same way as Line 5Ø. Move to the start point using BM, then set up the colour with C and the size with S.

LONGER MESSAGES

Defining strings for each message you want to display can become tedious if you have a large number of them.

The solution is to define all the characters you might want to use, by typing in a program like the one that follows:

```
10 PMODE 3,1
20 DIM LE$(26)
30 PCLS
40 FOR K=Ø TO 26:READ LE$(K):NEXT
50 FOR K=Ø TO 9:READ NU$(K):NEXT
60 DATA BR2,ND4R3D2NL3ND2BE2,ND4R3
   DGNL2FDNL3BU4BR2,NR3D4R3BU4BR2,
   ND4R2FD2GL2BE4BR,NR3D2NR2D2R3
   BU4BR2
70 DATA NR3D2NR2D2BE4BR,NR3D4R3U2
   LBE2BR,D4BR3U2NL3U2BR2,ND4BR2,
   BD4REU3L2R3BR2,D2ND2NF2E2BR2
80 DATA D4R3BU4BR2,ND4FREND4BR2,
   ND4F3DU4BR2,NR3D4R3U4BR2,ND4R3
   D2NL3BE2,NR3D4R3NHU4BR2
90 DATA ND4R3D2L2F2BU4BR2,BD4R3U2
   L3U2R3BR2,RND4RBR2,D4R2U4BR2,
   D3FEU3BR2,D4EFU4BR2
100 DATA DF2DBL2UE2UBR2,DFND2
   EUBR2,R3G3DR3BU4BR2
110 DATA NR2D4R2U4BR2,BDEND4BR2,
   R2D2L2D2R2BU4BR2,NR2BD2NR2BD2
   R2U4BR2,D2R2D2U4BR2,NR2D2R2D2L2
   BE4,D4R2U2L2BE2BR2,R2ND4BR2,
   NR2D4R2U2NL2U2BR2,NR2D2R2D2U4BR2
```

```
120 SCREEN1,Ø
130 A$="TESTING Ø123456789"
140 DRAW"BM6Ø,5Ø;C3S8"
150 GOSUB 9ØØØ
160 GOTO 16Ø
9ØØØ FOR K=1 TO LEN(A$)
9Ø1Ø B$=MID$(A$,K,1)
9Ø2Ø IF B$>="Ø" AND B$<="9"
   THEN DRAW NU$(VAL(B$)):GOTO 9Ø5Ø
9Ø3Ø IF B$="□" THEN N=Ø ELSE
   N=ASC(B$)-64
9Ø4Ø DRAW LE$(N)
9Ø5Ø NEXT
9Ø6Ø RETURN
```

A character set, consisting of capital letters from A to Z, digits from Ø to 9 and, most important, a space, is contained in the DATA lines—Lines 6Ø to 11Ø. This DATA is READ into the letter and number arrays—LE$ and NU$—by Lines 4Ø and 5Ø.

To use the character set, you must define the message, along with the start position and any other information, such as size and colour. Line 13Ø contains a test message, A$. In this case it is TESTING Ø123456789, but you could substitute anything you choose. Line 14Ø sets the start position, the colour and the size. With the message defined and the start point set, you can tell the machine to DRAW the message. The printing subroutine—starting at Line 9ØØØ—examines each character in A$, one at a time, finds the appropriate instructions in the arrays (see Games Programming, pages 144–147) and then DRAWs the character on the screen.

By changing the string in Line 13Ø, and, if necessary, the start position in Line 14Ø you can DRAW any message you wish. Since the part of the program that DRAWs the letters is a subroutine, you can have as many messages as you like during the program—you must call them all A$ though.

This is extremely useful if, for example, you have a series of messages you wish to display at various points during a game. You could incorporate the letter drawing program into the game as it stands—start the game at Line 12Ø or somewhere after the DATA, except for any DIM, CLEAR and PCLEAR lines which should be right at the start of the whole program. When you want to write the message, just set up strings containing the display you want, at the point it is needed in the program. Then add a DRAW statement like in Line 14Ø, and call the printing subroutine by including GOSUB 9ØØØ.

AND FINALLY

Once you've worked out the instructions for drawing a shape you can draw it the right way up, on either side, or even upside down. All you need to do is to include a further instruction in the string, A followed by a value from Ø to 3.

AØ means DRAW at Ø degrees—in other words upright. A1 will put the house on its side, leaning to the right—at 9Ø degrees. A2 will place the house upside down—at 18Ø degrees. And A3 will put the house on its side, leaning to the left—at 27Ø degrees.

Being able to control the angle at which the image is DRAWn is useful because you can DRAW a graphic in four different ways from just one set of instructions—you do not have to tell the computer how to DRAW each separate version of the graphic.

To see how it works in practice, type in and RUN this program:

```
1Ø PMODE 3,1
2Ø PCLS
3Ø SCREEN 1,Ø
4Ø S$="NR16E8F4U4R2D6F2D12L6U6L4
   D6L6U12"
5Ø FOR K=1 TO 2Ø
6Ø D=RND (2ØØ)+27:E=RND (14Ø)+27:
   C=RND (3)+1:A=RND (4)-1
7Ø DRAW "BM"+STR$(D)+","+
   STR$(E)+"C"+STR$(C)+"A"+
   STR$(A)+"XS$;"
8Ø NEXT K
9Ø GOTO 9Ø
```

You will find 20 houses, each identical except for their colour and orientation.

The shape of the houses is defined in the string in Line 4Ø. Four random numbers are chosen by Line 6Ø—D and E are the start coordinates, C is a colour and A is the angle at which the house will be DRAWn.

Line 7Ø DRAWs the house, adding together all the parts of the string and the random numbers. The STR$ function converts the numeric variables into strings—in effect it puts inverted commas round the value of the numeric variables. For example, if D=2 then STR$(D)="2".

What Line 7Ø does, then, is to begin with a blank move to a random start point, add a random colour instruction, and finally the DRAWing instructions themselves at a random angle. The X before S$ means 'execute a substring', and allows you to add one string to another during DRAWing rather like +HE$ did earlier.

The time to use X is when string space is limited in the machine's memory. X allows you to put a string together from various parts without actually creating a new string—adding strings together requires more string space, which wastes memory.

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

A

AND 35–36
Animation 26–32
Applications
- family finance 136–143
- hobbies' files 46–53, 75–79
- letter writer 124–128

Assembly language 66–67
Assignment statement 66–67, 92
ATTR, *Spectrum* 68–69

B

BASIC 65
BASIC programming
- arrays 152–155
- decision making 33–37
- how to PLOT, DRAW, LINE, PAINT 84–91
- inputting information 129–135
- programmer's road signs 60–64
- READ and DATA 104–109
- random numbers 2–7
- refining your graphics 184–192
- screen displays 117–123
- structured programming 173–178
- the FOR . . . NEXT loop 16–21
- variables 92–96

Binary 38, 41, 44, 45, 113–116
- negative numbers 179–183

Brackets, use of 35
Breaking out of a program 4
Bridge, drawing a
- *Spectrum* 108

Byte, definition of 114

C

Cassette recorders, choice of 24
Castle, drawing a
- *Dragon, Tandy* 108

Christmas program
- *Acorn* 64

CHR$, *Dragon, Tandy* 26–27
CIRCLE 86–91
CLEAR
- *Dragon, Tandy* 14, 27
- *Spectrum* 10

Clock, internal 69–73
COLOUR 87–90
Control variables 94
Craps program 63
Cursor, definition of 7
- control codes, *Commodores* 123

D

DATA
- for arrays 154–155
- for graphics 107–109
- machine code 67
- statements 8–14, 40–45

Decimal
- conversions from binary 38, 42
- converting fractions into binary 114

Decision making 33–37
Default colours
- *Acorn* 89
- *Dragon, Tandy* 90

Delays in programs 17
DIMensioning an array 152–153
DRAW 85–91
Drawing letters, *Dragon, Tandy* 191–192

E

Egg-timer program 176–177
ENDPROC, *Acorn* 64
Error, causes of 36

F

Family finance program 136–143
File, saving and loading a 77
Filing system program 46–53, 75–79
Flow charts 173–178
Flying bird sprite, *Commodore 64* 168–172
FOR . . . NEXT loop 16–21

G

Games
- aliens and missiles 144–151
- animation 26–32
- arrays for games 155
- bombing run program 161–167
- controlling movement 54–59
- firing missiles 55–58
- fruit machine 36
- guessing 3–5
- maze game 68–74
- minefield 97–103
- moving characters 54–59
- routines 8–15
- scoring and timing 69–73
- space station game 144–151
- visual explosions 161–167

GET, *Commodore 64* 55, 132–134
GET$, *Acorn* 55, 57, 58, 103, 132–134
GET#, *Commodore 64, Vic 20* 135
Golf-course, drawing a
- *Acorn, Spectrum* 184–191

GOSUB 62–64
GOTO 18–21, 60–62
Graph-drawing program
- *Acorn* 64

Graphics
- characters 38–45
- creating and moving UDGs 8–15
- drawing on the screen 132–133
- drawing pictures 107–109
- explosions for games 161–167
- fire-breathing dragon 80–83
- frog UDG 10–15
- instant embroidery 21
- low-resolution 26–32
- painting by numbers 19
- refining your graphics 184–192
- sunset pattern 20
- tank UDG 10–15
- using PLOT, DRAW, CIRCLE, LINE, PAINT 85–90
- also see animation; movement; ROM graphics; teletext; UDG.

H

Helicopter, building a
- *Commodore 64* 31

Hexadecimal 38, 42, 45, 156–160
HIRES, *Commodore 64* 87
Hobbies file 46–53, 75–79
House, drawing a
- *Acorn* 107–108
- *Commodore 64* 108–109

I

IF . . . THEN 3, 33–37
INK, *Spectrum* 86
INKEY, *Acorn* 28–29, 103, 134–135
INKEY$ 54–55, 132–135
INPUT 3–5, 117–122, 129–135
INPUT prompts 130–131
INT, *Commodore 64, Spectrum* 2–3

K

Keypress, detection of 54–55
Keywords, spelling of 19

L

Languages, computer 65
- see Assembly language; BASIC; Machine code

Letter writing program 124–128
LINE, *Dragon, Tandy* 88–91
Line numbers, in programs 7
Logical operators 35–37
Lower case letters,
- *Dragon, Tandy* 142

M

Machine code
- advantages of 66
- binary numbers 113–116
- drawing dragon with 80–83
- games graphics 38–45
- hexadecimal 156–160
- low level languages 65–67
- negative numbers 179–183
- nonary numbers 111–112
- number bases 110–116
- speeding up games routines 8–15

Maze programs 68–75
MID$, *Acorn* 71
- *Commodore 64* 101–102

Minefield game 97–99
MODE, *Acorn* 28
MOVE, *Acorn* 71, 88–90
Movement 26–32, 59

N

Negative binary numbers,
- conversion program 180–183

Nested loop, definition
- and use of 19

NEW 10–15, 23
Nonary numbers 111
Null strings 96
Number bases 110–116

O

ON . . . GOSUB 64
ON . . . GOTO 62
Opcodes 67
Operators 35
OR 35–36

P

Parameters 64
Password program 133
PAUSE
- *Commodore 64* 88
- *Spectrum* 101, 108

PEEK 59, 101
Peripherals, cassettes 22–25
Pixel 84
PLOT 88–89
PMODE, *Dragon, Tandy* 90
POINT, *Acorn* 71
POKE
- *Commodore 64* 15, 99, 108–109
- *Dragon, Tandy* 13, 40, 101
- *Spectrum* 101

Positioning text **117–123**
PRINT 26–32, 117–123
PRINT AT
- *Dragon, Tandy* 26–27
- *Spectrum, ZX81* 8–9, 31–32

PRINT TAB, *Acorn* 11, 28
- *Commodore 64, Vic 20* 30

PROCedures, *Acorn* 64
Program
- BASIC 8
- BREAKing into 4, 7, 11
- line numbers 7
- punctuation of 4
- slowing down 17

PSET, *Dragon, Tandy* 13, 90–91
Punctuation, in PRINT statements 119–123

R

RAM 25, 44, 46
Random numbers 2–7
READ 40–44, 104–109
REC, *Commodore 64* 87
Records 75–77
REPEAT . . . UNTIL, *Acorn* 36
Resolution, high and low 84
RESTORE 106–107
RETURN 62
RIGHT$, *Commodore 64* 101, 102
RND function 2–7
ROM graphics 26–32, 107–109
Running man, building a,
- *Acorn* 28–29

RUN/STOP, *Commodore 64, Vic 20* 7

S

Satellite, building a
- *Dragon* 26–27

SAVE 22–25
Scoring 97, 100–101
SCREEN, *Dragon, Tandy* 40, 90
Screen drawing program 132–133
Screen formatting 117–123
Shell, firing a 10–15
Ship, drawing a
- *Dragon, Tandy* 191

Simons' BASIC, *Commodore 64* 87–88
Snow scene, *Commodore 64* 186–188
Spaces, using
- *Commodore 64, Vic 20* 122

Sprite, *Commodore 64* 14, 15, 168–172
STEP 17, 21
STOP, *Spectrum, ZX81* 4, 64
String variables 4–5, 95–96
STRING$ 98
Subroutines 62–63
Symbols, arithmetic 6

T

TAB 117–122
Tables, multiplication 5–7
Tank, controlling and
- creating a 10–15

Teletext graphics, *BBC* 28
Terminating numbers 34
Timing 97, 101–103
Twos complement 179–183

U

UDG, definition of 8–15, 40–44
- grids for 8–11
- DATA for 45
- creating your own 38–45

V

VAL, *Commodore 64* 101
Variables 3–5, 92–96, 104–108
VDU command, *Acorn* 28–29, 70, 99
Verifying saved programs 24–25
VIC chip memory locations
- *Commodore 64* 172

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 7 ...

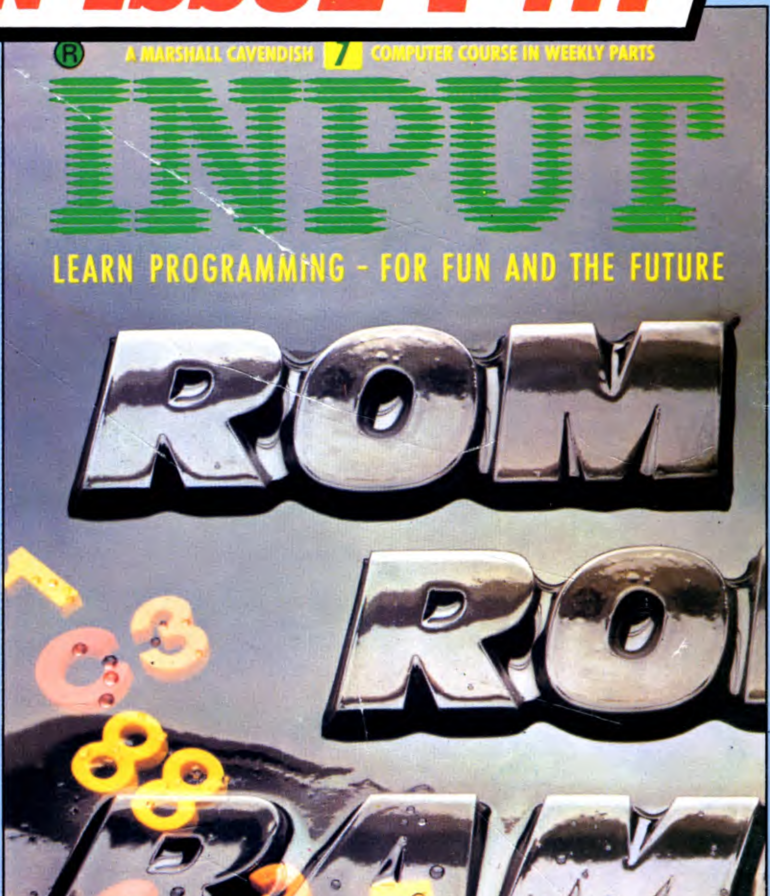
□ Discover how the MEMORY of your computer is built up—and what you are doing when you load machine code programs into it

□ In Games Programming, there's a run-down on how to add LEVELS OF DIFFICULTY to make your games suitable for beginners or experts—plus a complete new maze game program

□ Learn about STRING SLICING techniques which give you the ability to make use of stored information selectively

□ To bring together your skills at STRUCTURING A PROGRAM, there's an analysis of the techniques applied to a complete program for numeric or alphabetical sorting

□ JOYSTICKS are among the cheaper and more versatile of the peripherals you can add—and they're not just limited to games


®
A MARSHALL CAVENDISH 7 COMPUTER COURSE IN WEEKLY PARTS
INPUT
LEARN PROGRAMMING - FOR FUN AND THE FUTURE
ROM
RO
RAM

ASK YOUR NEWSAGENT FOR INPUT